Introducción a los sistemas digitales.

Un enfoque usando lenguajes de descripción de hardware.

José Daniel Muñoz Frías

© José Daniel Muñoz Frías.

Esta obra está bajo una licencia Reconocimiento – No comercial – Compartir bajo la misma licencia 2.5 España de Creative Commons. Para ver una copia de esta licencia, visite http://creativecommons.org/licenses/by-nc-sa/2.5/es/ o envíe una carta a Creative Commons, 171 Second Street, Suite 300, San Francisco, California 94105, USA.

Usted es libre de:

- copiar, distribuir y comunicar públicamente la obra.
- hacer obras derivadas.

Bajo las condiciones siguientes:

- **Reconocimiento.** Debe reconocer los créditos de la obra de la manera especificada por el autor o el licenciador (pero no de una manera que sugiera que tiene su apoyo o apoyan el uso que hace de su obra).
- **No comercial.** No puede utilizar esta obra para fines comerciales.
- **Compartir bajo la misma licencia.** Si altera o transforma esta obra, o genera una obra derivada, sólo puede distribuir la obra generada bajo una licencia idéntica a ésta.
- Al reutilizar o distribuir la obra, tiene que dejar bien claro los términos de la licencia de esta obra.
- Alguna de estas condiciones puede no aplicarse si se obtiene el permiso del titular de los derechos de autor.
- Nada en esta licencia menoscaba o restringe los derechos morales del autor.

Edición: *β*. Enero 2012.

A mis padres.

Índice general

Índice general v

1 Introducción **1**
1.1. Introducción a la técnica digital 1
1.2. Bits y niveles lógicos 4
1.3. Tecnologías para implantar circuitos digitales 7
1.4. Niveles de diseño 14
1.5. Ejercicios 17

2 Álgebra de Boole **19**
2.1. Definiciones y teoremas del álgebra de Boole 19
2.2. Funciones lógicas no básicas 23
2.3. Formas normales de una función booleana 24
2.4. Simplificación usando diagramas de Karnaugh 28
2.5. Ejercicios 33

3 Sistemas de numeración **37**
3.1. Introducción 37
3.2. Sistemas de numeración posicionales 38
3.3. Conversión entre bases 40
3.4. Rangos 42
3.5. Sistemas hexadecimal y octal 43
3.6. Operaciones matemáticas con números binarios 45
3.7. Representación de números enteros 48
3.8. Rangos en los números con signo 53
3.9. Operaciones matemáticas con números con signo 54
3.10. Otros códigos binarios 58
3.11. Ejercicios 63

4 Introducción al lenguaje VHDL **67**
4.1. Introducción 67
4.2. Flujo de diseño 68
4.3. Estructura del archivo 70
4.4. Ejemplos 74

4.5. Tipos de datos, constantes y operadores 78
4.6. Sentencias concurrentes 82
4.7. Ejercicios 86

5 Circuitos Aritméticos **87**
5.1. Sumador de un bit 87
5.2. Sumador de palabras de n bits 89
5.3. Restador de n bits 106
5.4. Sumador/Restador de n bits 107
5.5. Multiplicadores 108
5.6. Sumador de números en BCD natural 113
5.7. Ejercicios 115

6 Bloques Combinacionales **117**
6.1. Multiplexor 117
6.2. Demultiplexores 120
6.3. Codificadores 121
6.4. Decodificadores 127
6.5. Comparadores 127
6.6. Ejercicios 133

7 Circuitos secuenciales. Fundamentos **135**
7.1. Introducción 135
7.2. Conceptos básicos 136
7.3. Biestables 137
7.4. Ejercicios 146

8 Máquinas de estados finitos **149**
8.1. Introducción 149
8.2. Nomenclatura 149
8.3. Diseño de máquinas de estados 151
8.4. Descripción en VHDL 158
8.5. Detector de secuencia 165
8.6. Detector de secuencia usando detectores de flanco 169
8.7. Ejercicios 174

9 Registros **177**
9.1. Introducción 177
9.2. Registros de entrada y salida en paralelo 177
9.3. Registros de desplazamiento 180
9.4. Ejercicios 186

10 Contadores **189**
10.1. Introducción 189
10.2. Contador binario ascendente 189

10.3. Contador binario descendente 191
10.4. Contador ascendente / descendente 193
10.5. Contadores con habilitación de la cuenta 194
10.6. Contadores módulo m 196
10.7. Conexión de contadores en cascada 198
10.8. Contadores con carga paralelo 201
10.9. Contadores de secuencia arbitraria 203
10.10. Ejercicios 206

11 Diseño de sistemas complejos: ruta de datos + control 209
11.1. Introducción 209
11.2. Control de una barrera de aparcamiento 210
11.3. Control de calidad de toros 216
11.4. Conversor de binario a BCD 222
11.5. Interconexión de dispositivos mediante SPI 232
11.6. Ejercicios 241

Bibliografía 243

Índice alfabético 245

CAPÍTULO 1

Introducción

En este capítulo se realiza una introducción a los conceptos básicos que se utilizarán a lo largo del texto. Se expone la diferencia entre un sistema digital y uno analógico, remarcando las ventajas de los primeros frente a los segundos. A continuación se define qué es un **bit** y cómo se representa en un circuito. Se presentan a continuación algunas de las funciones básicas que se pueden realizar con los circuitos digitales y se termina el capítulo introduciendo los tipos de circuitos usados para diseñar sistemas digitales, haciendo especial énfasis en los circuitos digitales programables y el flujo de diseño seguido para configurarlos.

1.1. Introducción a la técnica digital

Una señal analógica es una representación análoga a la magnitud física que pretende sustituir. Por ejemplo si medimos con un osciloscopio la señal de salida de un micrófono, lo que veremos en la pantalla es una señal equivalente a la señal de audio que está capturando el micrófono. Así, el micrófono traduce las variaciones de presión del sonido capturado por su membrana en variaciones de tensión en sus conectores de salida.

En un sistema analógico, se usan componentes electrónicos para procesar señales analógicas, de forma que todo el circuito trabaja siempre con una copia analógica de la señal, tal como se ilustra en la figura 1.1, en la que se muestra un amplificador de audio.

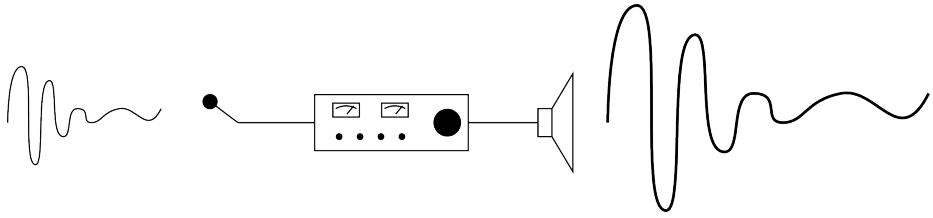


Figura 1.1: Sistema analógico: amplificador de audio.

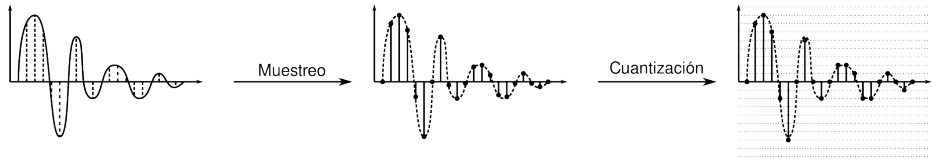


Figura 1.2: Muestreo y cuantización de una señal analógica

Las ventajas de este tipo de sistemas son:

- Las señales con las que trabaja el circuito electrónico son copias "exactas" de la señal analógica de entrada.
- Los circuitos para procesar las señales analógicas necesitan pocos transistores.
- Pueden manejar señales de potencia y de muy alta frecuencia.

Pero los inconvenientes son:

- Los circuitos analógicos son sensibles al ruido electromagnético, lo que hace que la señal deje de ser una copia exacta de la entrada. Se podría decir que la señal se "ensucia" conforme va avanzando por el circuito analógico.
- El comportamiento de los componentes electrónicos varía con la temperatura, edad del componente, etc. lo que hace que, para una misma señal de entrada, la salida del sistema analógico sea distinta de un circuito a otro, o cuando varía la temperatura.
- El conjunto de procesos que se puede realizar a una señal usando sistemas analógicos se limita a los muy sencillos. Ejemplos típicos son la amplificación (que no es más que multiplicar la señal de entrada por una constante) y el filtrado (que es una amplificación en función de la frecuencia de la señal).

1.1.1. Muestreo y cuantización

Para solucionar los inconvenientes de los sistemas analógicos se puede convertir una señal analógica en una secuencia de números, lo cual tiene muchas ventajas, como por ejemplo:

- Los números no pueden cambiarse por el ruido electromagnético ni por el envejecimiento de los componentes.
- Los números se pueden almacenar de forma fiable en una memoria.
- Se pueden realizar un sinfín de procesos a una señal representada en forma de una secuencia de números, como comprimir una señal de audio, reconocer voz, etc.

El proceso de conversión de una señal analógica en una secuencia de números consta de dos partes: muestreo y cuantización.

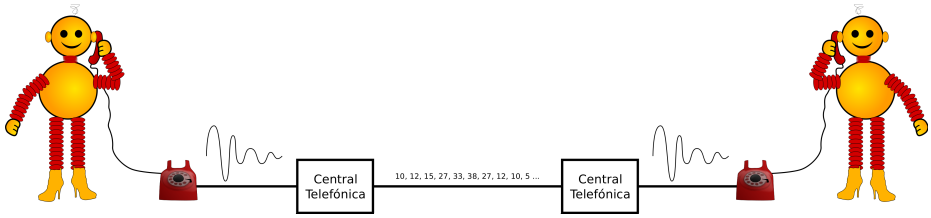


Figura 1.3: Conversión A/D y D/A en una conversación telefónica.

El muestreo de una señal analógica consiste en medir el valor de ésta a intervalos regulares de tiempo y quedarnos sólo con estas muestras; tal como se ilustra en la figura 1.2.

Según el teorema de muestreo de Nyquist-Shannon [Marks II, 1991], si la frecuencia de muestreo es dos veces mayor que la frecuencia máxima de la señal a muestrear, a partir de las muestras puede reconstruirse exactamente la señal analógica original.

En una señal muestreada, cada una de las muestras las medimos con precisión infinita. Si no disponemos de esa precisión, es necesario realizar un segundo proceso, denominado cuantización, en el que la cada muestra de la señal se representa con una precisión finita. Así en la figura 1.2 se muestra que el eje y sólo puede tener una serie de valores finitos, marcados por las líneas horizontales. Esta cuantización del eje vertical hace que haya que redondear los valores de las muestras al valor de la línea horizontal más próxima. El error cometido impide reconstruir exactamente la señal original a partir de sus muestras. No obstante, si se cuantizan las muestras con la suficiente precisión (haciendo que la distancia entre las líneas horizontales de la figura sea muy pequeña), el error de cuantización será despreciable y la señal reconstruida será prácticamente igual a la señal original.

El proceso de muestreo y cuantización se realiza en la práctica mediante un circuito denominado conversor analógico-digital (abreviado por conversor A/D). El proceso contrario de reconstrucción de la señal analógica a partir de sus muestras se realiza mediante un conversor digital-analógico (conversor D/A).

Aunque no se haya dado cuenta, en su vida diaria usa estos circuitos innumerables veces. Por ejemplo, cuando realiza una conversación con su teléfono fijo, en la centralita un conversor A/D muestrea y cuantifica su voz para convertirla en una secuencia de números que se envían por la red telefónica hasta la centralita del teléfono de destino. Allí, un conversor D/A convierte la secuencia de números de nuevo a una señal analógica que es escuchada por su interlocutor, tal como se ilustra en la figura 1.3.[1]

Realice el ejercicio 1

El interconectar digitalmente las centrales presenta numerosas ventajas:

[1] El que la conexión entre el teléfono y la centralita sea analógica es por razones históricas. Cuanto se digitalizó la red telefónica era más fácil cambiar las centralitas que cambiar los teléfonos de todas las casas.

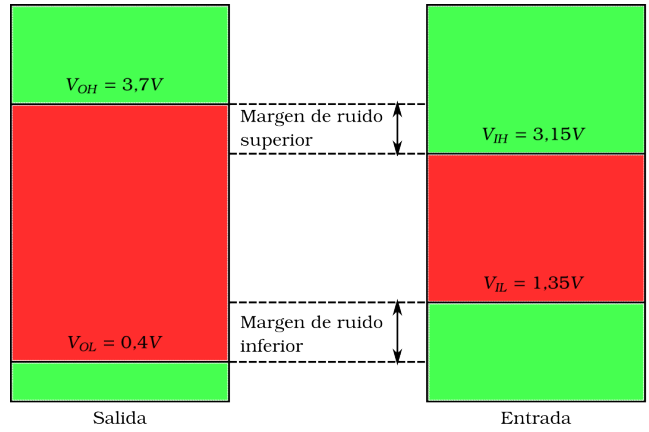


Figura 1.4: Niveles lógicos para la familia 74HC alimentada a 5 V.

- Las distancias entre centrales pueden ser mayores, al ser las señales digitales más inmunes al ruido
- Por la misma razón la calidad de la llamada es mayor.
- En una línea digital, gracias a la multiplexación, se pueden enviar varias llamadas por un solo cable. En el caso más simple, mediante un solo par de cobre (línea E1) pueden enviarse 30 llamadas. Si se usa fibra óptica, una sola fibra OC-192 puede llegar a transportar 150336 llamadas. Obviamente esto presenta un gran ahorro de costes para la compañía telefónica que se supone deben trasladar a los clientes.

1.2. Bits y niveles lógicos

En la sección anterior se ha mostrado que los sistemas digitales trabajan con números. La pregunta que se estará haciendo es cómo se puede "introducir" un número en un circuito y, sobre todo, cómo hacerlo de forma que dicho número no cambie debido a las perturbaciones que recibe el circuito. La respuesta es bien simple, en lugar de permitir que una señal pueda tomar todos los valores dentro del rango de la tensión de alimentación, se permite que la señal sólo pueda tomar dos valores: una tensión baja a la que normalmente se le asigna el valor 0 y una tensión alta a la que normalmente se le asigna el valor 1. Ahora bien, en lugar de definir una tensión única para representar el 0 y otra para representar el 1, se definen un rango de valores, tal como se muestra en la figura 1.4. Así, si la salida de un circuito digital ha de ser 0, la tensión ha de tener un valor entre 0 V y V_{OL} y si la salida ha de ser un 1, la tensión de salida ha de estar comprendida entre V_{OH} y V_{cc}, en donde V_{cc} es la tensión a la que se alimenta el circuito. De la misma forma, cualquier circuito digital cuya tensión de entrada esté entre 0 y V_{IL} interpretará que en dicha entrada

hay un 0 y si la tensión está entre V_{IH} y Vcc, interpretará que en dicha entrada hay un 1.

A la vista de lo anterior, conviene resaltar que:

- Como se puede apreciar en la figura 1.4 hay una diferencia entre V_{OL} y V_{IL}, a la que se denomina margen de ruido inferior y entre V_{OH} y V_{IH}, a la que se denomina margen de ruido superior. La razón de estos márgenes es conseguir que el circuito siga funcionando incluso en presencia de un ruido que perturbe la comunicación entre el circuito de salida y el de entrada.

- Los valores de V_{OL}, V_{OH}, V_{IL}, V_{IH} y V_{cc} dependen de la tecnología de fabricación del circuito. Existen en el mercado varias familias lógicas y cada una de ellas tiene una valor determinado para todas estas tensiones. Por ejemplo, en la figura 1.4 se han mostrado los valores para la familia lógica 74HC. Estos valores están publicados en las hojas de características de los dispositivos digitales.

- Ningún circuito que funcione correctamente tendrá en régimen permanente una tensión de salida entre V_{OL} y V_{OH}. Otra cosa son los transitorios, en los cuales obviamente la tensión ha de pasar por esa zona.

- No está definido qué hará un circuito si la tensión de entrada está comprendida entre V_{IL} y V_{IH}. Existe una tensión umbral, denominada V_{th}, en la cual el circuito deja de ver un 0 para pasar a ver un 1 en su entrada. El problema es que esta tensión umbral varía de unos dispositivos a otros, depende de la temperatura, de la tensión de alimentación,[2] de la edad del componente, etc. Por ello el fabricante del circuito lo único que garantiza es que V_{th} está comprendido entre V_{IL} y V_{IH}; es decir, que $V_{IL} < V_{th} < V_{IH}$.

No sólo de tensiones viven los circuitos

Además de los cuatro valores de tensión que acabamos de definir, existen otros cuatro parámetros de corriente asociados a éstos. Cuando la entrada de un circuito se somete a una tensión de nivel alto, ésta absorbe una determinada corriente, denominada i_{IH}. De la misma forma, cuando la entrada se somete a una tensión baja, la entrada cede una corriente denominada i_{IL}. Con las salidas ocurre lo mismo: cuando la salida se pone a nivel alto, el circuito es capaz de dar como máximo una corriente i_{OH} y cuando la salida se pone a nivel bajo el circuito es capaz de absorber una corriente i_{OL} como máximo. En la figura 1.5 se ilustra el sentido de estas corrientes y su valor para la familia 74HC.[3]

[2] Aunque la tensión ha de ser la marcada por el fabricante (V_{cc}), en la práctica es imposible generar una tensión de un valor exacto, produciéndose variaciones de unos milivoltios.

[3] Aunque no se ha mostrado en la figura, en las especificaciones del fabricante se indican con signo positivo las corrientes que salen del dispositivo y con signo negativo las que entran. Según esto, $i_{OH} = 4\,\text{mA}$ e $i_{OL} = -4\,\text{mA}$.

Figura 1.5: Corrientes en los circuitos digitales.

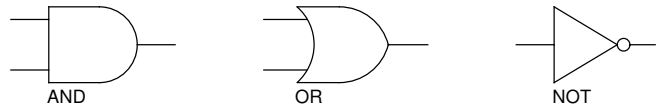


Figura 1.6: Puertas lógicas básicas

Bits y Bytes

A la vista de lo anterior, es posible diseñar un circuito que trabaje con un dígito, el cual sólo puede tomar los valores 0 y 1. A este dígito se le denomina dígito binario o **bit**, que es una contracción del inglés *Binary digIT*.

Con un sólo bit lo único que podemos expresar, aparte de los dos números 0 y 1, es el resultado de una expresión lógica que puede ser sólo cierta o falsa. Tradicionalmente se asigna 0 a falso y 1 a cierto. En el capítulo 2 demostraremos que cualquier expresión lógica puede realizarse usando tan sólo tres operaciones básicas: AND, OR y NOT. La gracia del asunto es que se pueden construir circuitos digitales que realizan también estas tres operaciones básicas. En la figura 1.6 se representan los símbolos de estos tres circuitos.

Si se desea trabajar con otro tipo de información distinta de los valores lógicos verdadero/falso, es necesario agrupar varios bits. Por ejemplo con dos bits tenemos cuatro combinaciones: 00, 01, 10 y 11. Por tanto con dos bits se puede representar una magnitud que tenga sólo cuatro posibles valores. Si se necesita un rango mayor de valores basta con añadir más bits.

A las agrupaciones de varios bits se les denomina **palabras** y la palabra más famosa es el **Byte**, que tiene 8 bits. Su popularidad radica en que es muy usada en informática porque como se verá en el capítulo 3, con 8 bits se pueden representar 256 valores y éstos son suficientes para representar las letras del alfabeto. Aunque en un sistema digital pueden usarse palabras de cualquier número de bits, en informática las palabras se restringen a múltiplos de 8 bits para que cualquier palabra pueda dividirse en Bytes. Así por ejemplo, en un PC pueden usarse datos de 8, 16, 32 y 64 bits.

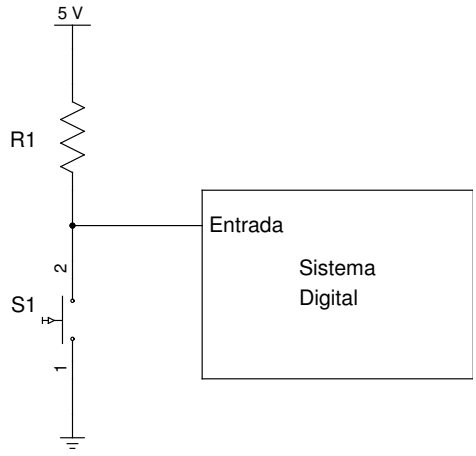
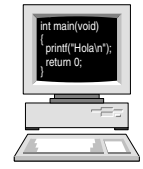


Figura 1.7: Circuito para generar un valor digital.

¿Cómo se representa un bit en un circuito?

En la figura 1.4 se ha mostrado que un circuito digital sólo puede tener una tensión de salida dentro de un rango de valores. Probablemente se esté preguntando cómo es posible hacer semejante cosa con un circuito electrónico. Si por el contrario no se lo ha preguntado, mal hecho, pues debería estar ansioso por saber cómo hacerlo.

En la figura1.7 se muestra un circuito usado para introducir un bit en un sistema digital alimentado a 5 V. El circuito funciona de la siguiente manera: cuando se accione el pulsador S1, la entrada del circuito digital se pondrá a cero. Cuando no se accione, la tensión de la entrada será igual a 5 V. En ambos casos se han supuesto componentes ideales, es decir, un pulsador con resistencia igual a cero y un sistema digital con una corriente de entrada 0. En un sistema real las tensiones serán algo distintas, pero recuerde que una ventaja de los sistemas digitales es su inmunidad a este tipo de problemas. Si observa la figura 1.4 verá que tenemos un gran margen de maniobra para conseguir que el circuito funcione correctamente.



Realice los ejercicios 2, 3 y 4

1.3. Tecnologías para implantar circuitos digitales

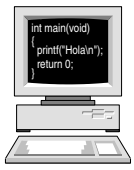
Los circuitos digitales se construyen a partir de puertas lógicas como las que se han mostrado en la figura 1.6. A lo largo de la historia se han usado distintas tecnologías para implementar estas puertas. Las primeras puertas lógicas electrónicas se construyeron usando válvulas de vacío. Un ejemplo clásico de este tipo de tecnología es el primer ordenador electrónico, llamado ENIAC,[4] construido en 1946 en

[4] De: Electronic Numerical Integrator And Computer.

la Universidad de Pennsylvania [Goldstine and Goldstine, 1996]. En la figura 1.8(a) se muestra uno de sus circuitos sujetado por dos de sus creadores; lo cual da una idea clara de su tamaño. No obstante, a pesar del tamaño, el circuito mostrado es más bien simple: permite almacenar un dígito decimal. Por otro lado hay que destacar que aunque este primer ordenador fue una revolución en su época, el hecho de que la tecnología usada para implantarlo era muy primitiva hizo que sus prestaciones fuesen de lo más modestas para los estándares actuales:

- Tamaño: 2,6 m de alto x 0,9 m de ancho x 24 m de largo, ocupando una superficie de 63 m^2. Vamos, un auténtico mastodonte.[5]
- Potencia consumida: 150 kW.
- Peso: 27 Tm.
- Frecuencia de reloj: 100 kHz.
- Prestaciones: 5000 sumas por segundo.

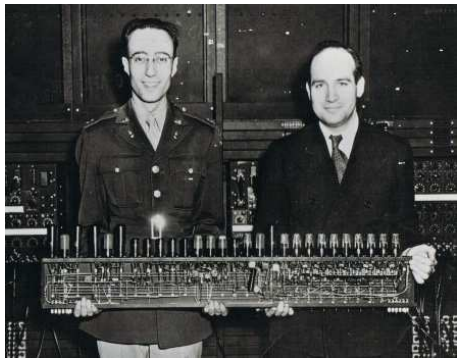
Con el desarrollo del transistor se desarrollaron puertas lógicas basadas en transistores, diodos y resistencias. En la figura 1.8(b) se muestra un módulo del ordenador IBM 1401, construido en el año 1959. Dicho módulo contiene 4 puertas lógicas y su tamaño, peso y consumo es mucho menor a su equivalente con válvulas. El siguiente paso fue integrar todos estos componentes en un sólo circuito integrado. En la figura 1.8(c) se muestra un circuito integrado que contiene 4 puertas lógicas, con un tamaño mucho menor que el equivalente con componentes discretos. En cuanto aparecieron estos primeros chips, denominados SSI (*Small Scale Integration*) pues contenían unos pocos transistores; los diseñadores de ordenadores empezaron a usarlos. Así, un ejemplo notable de ordenador basado en chips SSI es el AGC (*Apollo Guidance Computer*), diseñado en 1960 para controlar la navegación de los cohetes Apollo que llevaron al hombre a la luna [Wikipedia, 2011]. A finales de los años 60, con la mejora de las tecnologías de fabricación se conseguían integrar cientos de transistores en un solo chip, dando lugar a la tecnología MSI (*Medium Scale Integration*). Estos chips eran capaces de contener circuitos digitales básicos, como sumadores, contadores, etc. Un ejemplo de ordenador construido con este tipo de tecnología es el Computer Automation Inc. LSI-2, que fue el primer ordenador que se instaló en el ICAI y que puede verlo en el descansillo de la primera planta. El desarrollo de la tecnología ha permitido integrar cada vez más componentes en un solo circuito integrado, sucediéndose las generaciones de chips: LSI (*Large Scale Integration*) con decenas de miles de transistores, VLSI (*Very Large Scale Integration*) con cientos de miles de transistores y ULSI (*Ultra Large Scale Integration*) con millones de transistores. Un ejemplo de esta última generación de chips se muestra en la figura 1.8(d). El chip mostrado es un microprocesador Intel Core i7–2600 que contiene 995 millones de transistores.[6]



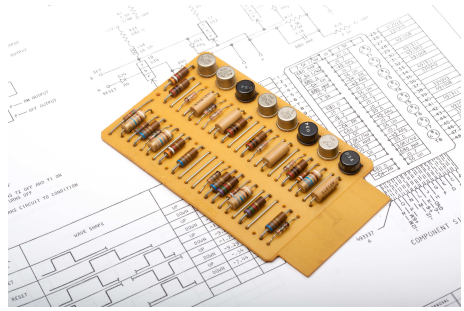
Realice el ejercicio 5

[5] El ordenador se disponía en una serie de chasis en forma de U.

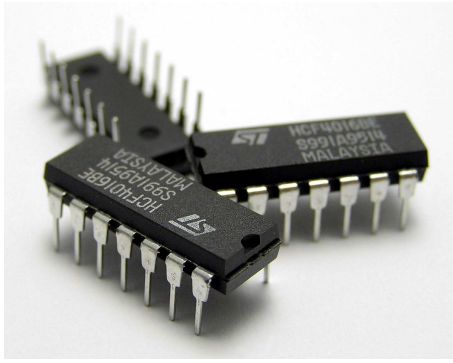
[6] El procesador incluye 4 núcleos, 8 MB de memoria caché, controlador de memoria y controlador gráfico HD; todo ello en un chip de 216 mm^2.



(a) Válvulas de vacío



(b) Transistores discretos



(c) Circuito integrado SSI

(d) Circuito integrado ULSI

Figura 1.8: Distintas tecnologías para implantar circuitos digitales.

Ley de Moore

El crecimiento en la integración de circuitos ya fue predicho por uno de los fundadores de Intel, Gordon Moore, en el año 1965. Viendo la tendencia pasada en la integración de circuitos predijo que el número de transistores que se podrían integrar en un chip se incrementaría cada dos años. No se trata de una ley física, sino de una observación estadística que se ha mantenido hasta hoy gracias a las continuas inversiones de la industria de semiconductores en I+D; las cuales han permitido mejorar continuamente los procesos de fabricación.

Un ejemplo de esta mejora lo tenemos en la propia Intel, que dicho sea de paso fueron los creadores del primer microprocesador. Éste, denominado el 4004, fue fabricado en 1971 con una tecnología de 10μm (10,000nm) y constaba de 2300 transistores.[7] Un procesador actual como el Intel Core i7-2600 se está fabricando con

[7] Las tecnologías de fabricación de circuitos integrados se caracterizan por la dimensión mínima

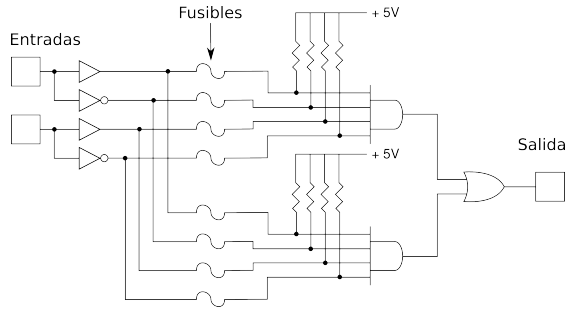


Figura 1.9: *Programmable Array Logic*.

tecnología de 32nm y como se ha mencionado anteriormente contiene 995 millones de transistores.

Otro ejemplo curioso que muestra el avance de la tecnología es el modo en que un grupo de estudiantes de la Universidad de Pennsylvania celebraron el 50 aniversario del ENIAC: construir una réplica del ENIAC en un solo chip [Van der Spiegel, 1995]. Así, los 63 m^2 que ocupaba el ENIAC original se redujeron a un bloque de silicio de 0,5 mm de espesor y de 7.44 x 5.29 mm de superficie ($39,35 \cdot 10^{-6} m^2$) , lo cual, todo hay que decirlo, es un gran avance.

1.3.1. Lógica programable

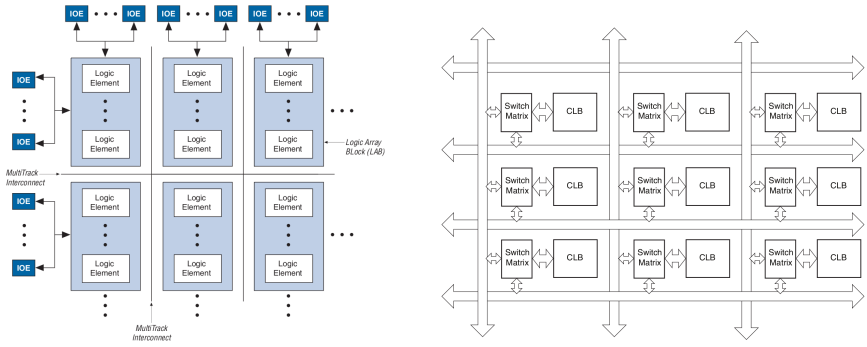
Aunque los primeros circuitos integrados se diseñaban con una función fija: unas cuantas puertas lógicas, un microprocesador, una memoria, etc. a finales de los años 70 se empezaron a desarrollar circuitos cuya funcionalidad podía ser configurada por el usuario. A este tipo de circuitos se les denomina con el nombre genérico de lógica programable.[8]

Los primeros dispositivos programables, denominados PAL (*Programmable Array Logic*) fueron introducidos por MMI (*Monolithic Memories, Inc.*) en 1978. Su estructura es la mostrada en la figura 1.9. Como veremos en el capítulo 2, cualquier función lógica puede implantarse mediante dos niveles de puertas AND-OR. Por ello, estos circuitos ofrecían una serie de puertas AND que al fundir los fusibles adecuados quedaban conectadas a una entrada o a su negada; pudiendo configurarse así cualquier función lógica. Este tipo de circuitos se popularizó rápidamente, pues permitían integrar en un solo chip el equivalente a varios circuitos SSI. Además si ocurría un fallo de diseño no era necesario hacer un circuito nuevo; bastaba con reprogramar la PAL.

---

de los componentes que pueden integrarse en el chip.

[8] En este contexto por programable se entiende que existen en el circuito una serie de puertas lógicas cuya conexión puede ser configurada por el usuario. Es importante distinguir este uso del término programable con el que se le da en el ámbito del software, en el que se dice que un ordenador es programable porque se le dan una serie de instrucciones para que las ejecute secuencialmente.



(a) CPLD MAX V de Altera. (b) FPGA Spartan 6 de Xilinx.

Figura 1.10: Diagramas de bloques simplificados de una CPLD y una FPGA.

Con el avance de las tecnologías de integración, las PAL fueron creciendo en densidad y complejidad, dando lugar a dos familias de dispositivos: CPLD (*Complex Programmable Logic Devices*) y FPGA (*Field Programmable Gate Array*). Las CPLD no son más que una colección de PALs interconectadas internamente mediante una matriz de conexiones que también es programable, tal como se observa en la figura 1.10(a). Por el contrario las FPGA contienen una serie de bloques lógicos elementales interconectados, tal como se muestra en la figura 1.10(b). Dichos bloques lógicos son más simples que una PAL, pudiendo implementar típicamente una función lógica de cuatro entradas y almacenar un bit. Aunque a primera vista puede parecer que ambos dispositivos son lo mismo, la diferencia radica en que la unidad mínima interconectada en la FPGA es más simple que en la CPLD, dando lugar a una estructura más flexible. Las diferencias más notables entre ambos tipos de dispositivos son las siguientes:

1. Las FPGA contienen más lógica que los CPLD. Así mientras un CPLD puede contener del orden de mil elementos lógicos,[9] una FPGA puede contener del orden de un millón. Por ejemplo, el fabricante Altera ofrece como tope de gama en CPLD el dispositivo MAX V 5M2210Z con 2.210 elementos lógicos. El tope de gama en FPGA es el dispositivo Stratix V 5SGXAB con 1.052.000 elementos lógicos.

2. La configuración de los CPLD es no volátil, es decir, no se pierde cuando se apaga el dispositivo. Por el contrario las FPGA por lo general guardan su configuración en células internas de memoria RAM que desaparecen al desconectar la alimentación. Esto obliga a reconfigurar las FPGA cada vez que se arranca el circuito, existiendo para ello chips externos de memoria no volátil para almacenar dicha configuración.

[9] Un elemento lógico es el bloque mínimo

3. Al ser dispositivos más densos, las FPGA suelen incluir, además de los bloques lógicos elementales, una serie de bloques con funcionalidad fija. Por ejemplo la familia Cyclone II de Altera contiene multiplicadores de 18x18 bits y 29 kB de memoria RAM.

Microcontroladores

Tanto las CPLD como las FPGA son circuitos que se pueden configurar para contener un circuito lógico formado por miles de puertas. En este caso la "inteligencia" del circuito está formada precisamente por la forma de interconectar esas miles de puertas. La otra alternativa para darle "inteligencia" a un circuito es mediante un microprocesador en el que se ejecuta un programa, que es donde reside ahora dicha "inteligencia". Al primer modelo se le denomina lógica cableada, pues el comportamiento del circuito se define por cómo se cablean sus componentes. Al segundo modelo se le denomina lógica programada, pues el comportamiento del sistema se define por el *software* que ejecuta el microprocesador.

El primer ejemplo de circuito con lógica programada fue una calculadora de la ya difunta empresa japonesa Busicom. Dicha calculadora estaba basada en el microprocesador Intel 4004 mencionado anteriormente. No obstante el 4004 necesitaba de varios chips a su alrededor para funcionar: una memoria ROM para el programa, una RAM para los datos y una serie de chips de entrada/salida para interactuar con el exterior [Augarten, 1983]. Un poco después (1974) Texas Instruments sacó al mercado el TMS1000, que es considerado el primer microcontrolador. La diferencia con el 4004 es que integraba en el mismo chip un microprocesador, una ROM de 1 Kbit, una RAM de 256 bits y la circuitería de entrada/salida. Por tanto, a diferencia del 4004, podemos darle "inteligencia" a un circuito sin más que añadir un solo chip.

Los microcontroladores modernos son mucho más potentes que estos chips pioneros, pero la filosofía sigue siendo la misma. En la figura 1.11 se muestra el diagrama de bloques del microcontrolador ATmega328, que es el usado en las placas Arduino, muy populares entre los aficionados a la electrónica. Como puede observar el microcontrolador incluye una CPU con una memoria FLASH para almacenar el programa y una memoria RAM para los datos. El resto de elementos son circuitos de soporte de la CPU, como circuitería de programación, reloj, etc. y circuitos para facilitar la entrada/salida. Así, dispone de un conversor A/D, temporizadores, circuitos de comunicaciones y puertos de entrada/salida digitales.

1.3.2. Tecnología usada en la actualidad

En la actualidad la mayoría de los diseños digitales ya no usan los circuitos SSI y MSI, sino que usan lógica programable. El qué tipo de circuito programable elegir depende en primer lugar de la velocidad que es necesaria en el sistema. Así, si las señales no son de alta velocidad lo más interesante, tanto desde el punto de vista económico como de facilidad de diseño, es usar un microcontrolador. Si por el contrario el sistema necesita controlar señales de alta velocidad, no hay más

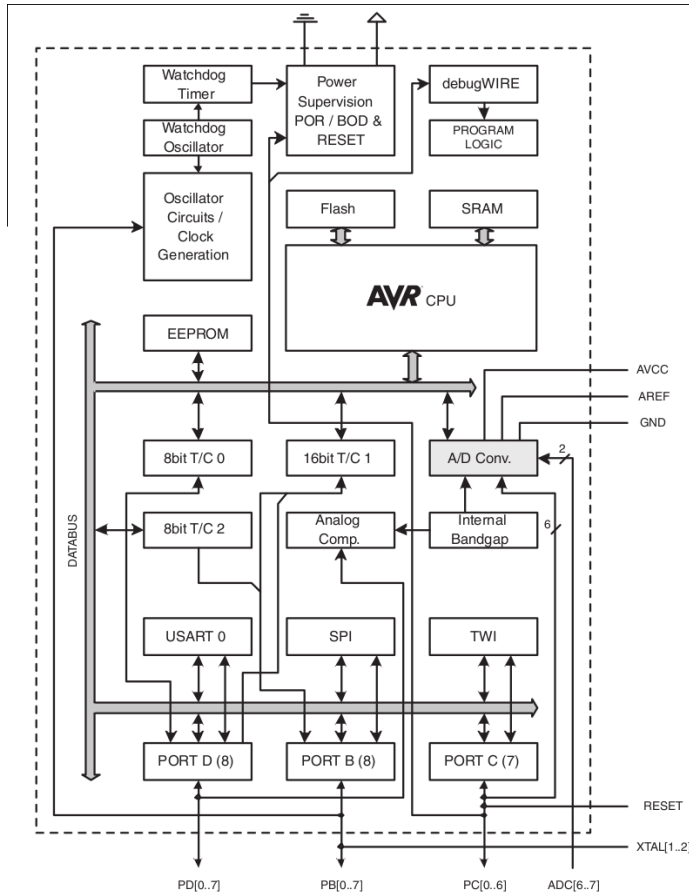


Figura 1.11: *Diagrama de bloques del microcontrolador ATmega328.*

remedio que usar lógica cableada. En estos casos se elige una CPLD si el diseño no es muy complejo o una FPGA en caso contrario.

Las ventajas de usar lógica programable frente a lógica discreta son muchas:

- Flexibilidad: al ser elementos reconfigurables, cualquier cambio en el diseño no necesita cambiar el circuito físico, sino simplemente reconfigurar los dispositivos programables.
- Menor espacio: al poder integrar millones de puertas en un solo chip el espacio ocupado en la placa de circuito impreso es mucho menor.
- Mayor velocidad: al estar todos los componentes situados en el mismo chip, las señales pueden viajar más rápido entre ellos.
- Menor coste: al reducirse el tamaño de la placa de circuito impreso, esta será

más barata (y también la caja). Además un solo chip es más barato que los chips SSI y MSI que sustituye y consume menos energía.

Por último, cuando de un circuito se van a fabricar cientos de miles de unidades o cuando se necesita una muy alta velocidad, es necesario recurrir a chips diseñados a medida, denominados ASIC (*Application Specific Integrated Circuit*). Este tipo de chips se diseñan con una función fija que no se puede cambiar y su principal problema son los elevados costes no retornables (costes de preparación de máscaras, prototipos, bancos de prueba, etc.) que para un chip de un millón de puertas pueden estar en el orden de medio millón de Euros. Al tener que repartir estos costes en los chips vendidos, es necesario vender muchos para que el precio sea competitivo frente a una solución basada en lógica programable. Por otro lado, si ocurre un fallo en el diseño o cambian las especificaciones es necesario fabricar un nuevo chip, con lo cual es necesario volver a pagar los costes no retornables.

1.4. Niveles de diseño

El diseño de circuitos digitales, al igual que otras disciplinas, puede realizarse a distintos niveles de abstracción. En general cuanto más bajamos de nivel, más control tenemos sobre el circuito realizado y por tanto se pueden realizar circuitos más óptimos. El problema es que según bajamos de nivel es necesario un mayor esfuerzo para diseñar los circuitos; por lo que para diseñar circuitos con millones de transistores es necesario trabajar en niveles de abstracción muy altos. En esta sección se exponen los distintos niveles de diseño de circuitos digitales.

1.4.1. Técnicas de fabricación y física de semiconductores

El nivel más bajo es el que estudia las técnicas de fabricación de los circuitos integrados. Ciertamente el trabajo de los ingenieros "situados" en este nivel ha conseguido que se siga cumpliendo la ley de Moore desde que se enunció en los años 60. En este nivel se estudia cómo conseguir fabricar transistores más pequeños y más rápidos, pero no se pasa de ahí. Es decir, una vez que se han diseñado los componentes básicos (los transistores, resistencias, etc.) se pasa al nivel superior de abstracción. En este curso no volveremos a hablar de este nivel, pero es conveniente que tenga en cuenta su importancia.

1.4.2. Nivel de transistor

Donde termina el nivel anterior, en el diseño de un transistor, comienza este nivel. Los ingenieros que trabajan en este nivel diseñan componentes a partir de transistores. Estos componentes son muy básicos: puertas lógicas, elementos de memoria, etc. Cada uno de estos elementos se compone de unos cuantos transistores de forma que su diseño y optimización es posible en un tiempo razonable.[10]

[10] Los primeros circuitos integrados se diseñaron en este nivel. Por ejemplo el microprocesador 4004 de Intel se diseñó íntegramente con transistores, como se puede ver en sus planos [Intel, 1973].

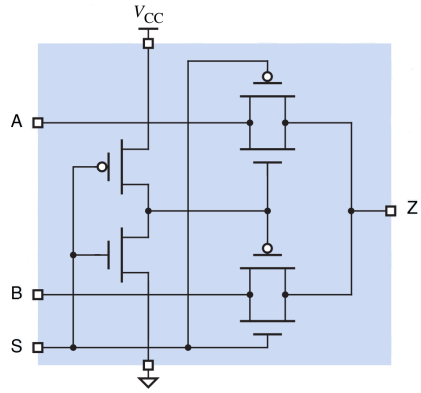


Figura 1.12: *Multiplexor diseñado a nivel de transistor.*

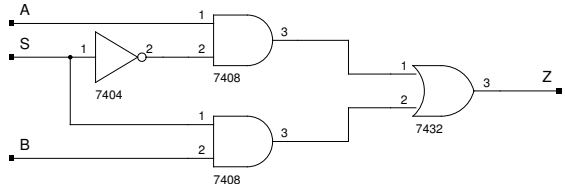


Figura 1.13: *Multiplexor diseñado a nivel de puerta.*

Un ejemplo de diseño a este nivel lo podemos observar en la figura 1.12. Este circuito es un multiplexor y funciona de la siguiente manera: cuando la entrada de control S vale 0 la salida Z es igual a la entrada A y cuando la entrada S es igual a 1, la salida Z es igual a la entrada B. La utilidad de este circuito se verá a lo largo del curso, pero en esta introducción nos servirá para ilustrar cómo varía la forma de describir un mismo circuito en función del nivel de diseño en el que se trabaja.

1.4.3. Nivel de puerta

En este nivel los circuitos se construyen usando puertas lógicas en lugar de transistores. Por ejemplo, en la figura 1.13 se muestra el mismo multiplexor mostrado en el apartado anterior pero diseñado a nivel de puerta. Para demostrarlo, basta con observar que el circuito implementa la función lógica $Z = \overline{S} \cdot A + S \cdot B$. Por tanto, cuando $S = 0$ la ecuación anterior queda como $Z = 1 \cdot A + 0 \cdot B = A$ y de la misma forma cuando $S = 1$ la salida $Z = B$.

Es fácil darse cuenta que en este nivel el diseño de circuitos es más fácil que en el nivel de transistores. No obstante, en ocasiones los circuitos pueden ser menos óptimos. Por ejemplo en este caso el multiplexor necesita 14 transistores,[11] mientras

[11] Cada puerta AND y OR necesita 4 transistores y la puerta NOT 2.

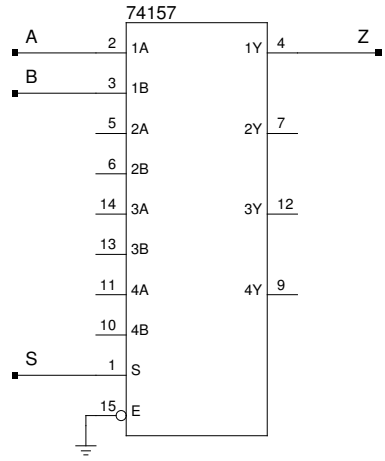


Figura 1.14: *Multiplexor diseñado a nivel de bloque.*

que en el diseñado a nivel de transistor se usaron 6.

1.4.4. Nivel de bloque

En el diseño digital hay bloques que se usan en muchos diseños. Uno de ellos es precisamente el multiplexor. Por ello los fabricantes de circuitos tienen en su catálogo chips MSI que implementan estos bloques. Por ejemplo el circuito 74HC157 contiene 4 multiplexores de dos entradas, por lo que para construir el multiplexor que hemos diseñado en las secciones anteriores sólo tenemos que conectar las entradas y salidas al chip, tal como se muestra en la figura 1.14.

Al usar como elementos constructivos del circuito digital bloques con una funcionalidad más compleja que las puertas lógicas elementales, el diseño se realiza de forma mucho más cómodo y rápido.

1.4.5. Diseño con lenguajes de descripción de hardware

El nivel más alto de abstracción, que es el usado hoy en día para los diseños complejos, es el uso de lenguajes de descripción de *hardware*, conocidos por sus siglas en inglés HDL (*Hardware Description Language*). Mediante estos lenguajes se describe el circuito mediante un fichero de texto. Este fichero puede ser usado como entrada para varias herramientas CAD. Las más habituales son:

- Simuladores. Estos programas toman como entrada una descripción VHDL de un circuito y son capaces de simular el funcionamiento del mismo. Esto nos permite verificar el funcionamiento del circuito antes de fabricarlo o de descargarlo en un circuito de lógica programable.
- Sintetizadores. Estos programas generan un circuito digital a partir de una descripción en HDL. La salida de estos programas es un archivo que se puede

descargar en un dispositivo programable o enviar a un fabricante para que nos construya un ASIC (si podemos pagarlo, claro está).

En este texto vamos a usar el lenguaje VHDL para describir circuitos digitales. Aunque es un lenguaje muy complejo y potente, no es difícil describir circuitos sencillos con el. Por ejemplo, el multiplexor de dos entradas diseñado en los apartados anteriores puede especificarse en VHDL de la siguiente manera:

```
library IEEE;
use IEEE.std_logic_1164.all;

entity Mux2 is
  port( a, b, s: in  std_logic;
        z      : out std_logic );
end Mux2;

architecture RTL of Mux2 is
begin
  z <= a when s = '0' else b;
end RTL;
```

Las primeras dos líneas definen una serie de librerías usadas por el sintetizador y el simulador. Piense en ellas como una especie de `#include <stdio.h>` de C. Las siguientes 4 líneas definen las entradas y salidas del circuito, que en la terminología de VHDL es una "entity". En este caso se definen como entradas las señales `a`, `b` y `s` y como salida la señal `z`. Por último, las 4 últimas líneas definen el comportamiento del circuito, que en la terminología de VHDL es la "arquitectura" del circuito. En este caso se está especificando que la salida `z` ha de ser igual a la señal `a` cuando `s` valga 0 o a la señal `b` cuando `s` valga 1.

Como conclusión, conforme hemos ido subiendo de nivel, el mismo circuito se ha ido especificando de manera más compacta, lo que aumenta la productividad del diseñador y hace posible que circuitos con miles de millones de transistores se puedan diseñar en un tiempo razonable.

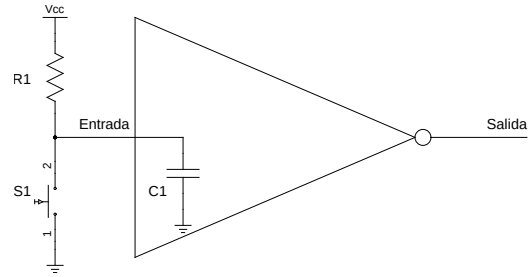
1.5. Ejercicios

1. En la figura 1.3 se muestra el proceso de conversión A/D y D/A realizado en los sistemas de telefonía fija. Para ahorrar ancho de banda, el teléfono elimina las frecuencias de audio superiores a 4 kHz.[12] ¿A qué frecuencia ha de muestrearse el sonido telefónico en la centralita? Teniendo en cuenta que cada muestra se cuantifica con 8 bits, calcule el ancho de banda necesario para una conversación, expresándolo en bits por segundo.
2. En la figura 1.7 se muestra un circuito de entrada para un sistema digital. Calcule el valor máximo que puede tener la resistencia R1 para que el circuito

[12] Esta es la razón por la que el sonido telefónico no es de muy alta fidelidad que se diga.

funcione correctamente. Para ello suponga una $i_{IH} = 1\,\mu$A y que $V_{IH} = 3,15\,$V. **Pista:** Cuando el pulsador esté en reposo, la tensión de entrada del circuito ha de estar dentro del rango definido para el 1.

3. Con la resistencia calculada en el ejercicio anterior y suponiendo una capacidad parásita a la entrada del circuito digital de 10 pF, calcule el tiempo que tardará la señal de entrada en pasar de 0 a 5 V.



4. Teniendo en cuenta que en un circuito RC como el de la figura anterior, la tensión en bornes del condensador viene dada por la expresión $V_c(t) = V_{cc} \cdot (1 - e^{-t/R \cdot C})$, calcule el tiempo que tardará la salida de la puerta en pasar de 1 a 0 cuando se suelte el pulsador. Recuerde que la puerta conmuta su salida cuando su entrada es igual a la tensión umbral V_{th}. Suponga que en la familia lógica usada en el circuito la tensión umbral es igual a la mitad de la tensión de alimentación.
5. Consulte la documentación sobre el Apollo Guidance Computer en la web y compare sus prestaciones con las del ENIAC.
6. Defina los siguientes acrónimos: ASIC, CAD, CD, DVD, CPLD, FPGA, HDL, IC, SSI, LSI, MSI, VLSI, ULSI, DIP, SMD, PCB, VHDL, MPEG.

CAPÍTULO 2

Álgebra de Boole

En electrónica digital se trabaja con variables que pueden tener sólo dos valores, 0 ó 1. Por suerte los primeros ingenieros que trabajaron con electrónica digital ya encontraron las matemáticas necesarias desarrolladas, pues en 1854, mucho antes de que se hubiera inventado el transistor, George Boole definió un álgebra para manipular variables que sólo podían tener los valores cierto y falso, lo cual permitía según Boole investigar las leyes del pensamiento [Boole, 1854]. En este capítulo se introducen los conceptos básicos de este álgebra.

2.1. Definiciones y teoremas del álgebra de Boole

2.1.1. Definiciones

El álgebra de Boole se basa en las siguientes definiciones:

Conjunto de valores de una variable

Una variable Booleana sólo puede tomar dos valores:

$$x = 0 \quad \text{ó} \quad x = 1 \tag{2.1}$$

Operación NOT

La operación NOT, también llamada complemento o negación se define como:

$$\begin{aligned} \overline{0} &= 1 \\ \overline{1} &= 0 \end{aligned} \tag{2.2}$$

Operación AND

La operación AND, también llamada producto lógico o booleano se define como:

$$\begin{aligned} 0 \cdot 0 &= 0 \\ 0 \cdot 1 &= 0 \\ 1 \cdot 0 &= 0 \\ 1 \cdot 1 &= 1 \end{aligned} \tag{2.3}$$

Operación OR

La operación OR, también llamada suma lógica o booleana se define como:

$$\begin{aligned} 0 + 0 &= 0 \\ 0 + 1 &= 1 \\ 1 + 0 &= 1 \\ 1 + 1 &= 1 \end{aligned} \tag{2.4}$$

Nótese que la suma lógica se diferencia de la suma de números naturales sólo en la última fila, es decir $1 + 1 = 1$. Esto dará lugar a que aunque la mayoría de los teoremas del álgebra de Boole son equivalentes a los del álgebra elemental, hay ciertos teoremas que sólo se cumplen en el álgebra de Boole, lo cual puede dar lugar a confusiones al manipular expresiones booleanas.

2.1.2. Teoremas

Los siguientes teoremas se demuestran a partir de las definiciones anteriores. Las demostraciones se pueden realizar manipulando las expresiones o por inducción perfecta. Este último método consiste en comprobar el teorema para todas las posibles combinaciones de las variables, dándolo por válido si se cumple en todos los casos. Esto que para los números naturales sería imposible al tener infinitos valores, para las variables booleanas es factible al tener sólo los valores 0 ó 1. Ilustraremos esto último con la demostración del primer teorema.

Por otro lado, veremos que en el álgebra de Boole todos los teoremas tienen una versión dual, que es la obtenida cambiando los ceros por unos, los productos por sumas y viceversa.

Elemento identidad

$$0 + x = x \qquad 1 \cdot x = x \tag{2.5}$$

Para demostrar la primera ecuación se puede aplicar inducción perfecta. Teniendo en cuenta la definición de la operación OR (ecuación 2.4):

x	$0+x$
0	0
1	1

Por tanto, como para todos los posibles valores de x, $0 + x = x$, el teorema queda demostrado.

Elemento nulo

$$x + 1 = 1 \qquad x \cdot 0 = 0 \tag{2.6}$$

Nótese que este teorema aparece la primera diferencia con el álgebra elemental: si a cualquier variable le sumamos 1, el resultado es 1 independientemente del valor de la variable.

Propiedad conmutativa

$$x + y = y + x \qquad x \cdot y = y \cdot x \tag{2.7}$$

Para demostrarlo podemos aplicar inducción perfecta:

x	y	$x+y$	$y+x$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

Para calcular $x+y$ e $y+x$ se ha usado la operación OR definida en la definición 2.4. Como ambas operaciones han dado el mismo resultado para todas las combinaciones posibles de variables de entrada, el teorema queda demostrado.

Propiedad distributiva

$$x \cdot (y + z) = x \cdot y + x \cdot z \qquad x + (y \cdot z) = (x + y) \cdot (x + z) \tag{2.8}$$

En primer lugar cabe destacar que al igual que en el álgebra elemental, el producto booleano tiene precedencia sobre la suma booleana, por lo que $x \cdot y + x \cdot z$ es equivalente a $(x \cdot y) + (x \cdot z)$.

En segundo lugar nótese que al contrario que en el álgebra elemental, en el álgebra de Boole también existe la propiedad distributiva respecto de la suma.

Elemento complementario

$$x + \overline{x} = 1 \qquad x \cdot \overline{x} = 0 \tag{2.9}$$

Idempotencia

$$x + x = x \qquad x \cdot x = x \tag{2.10}$$

Convolución

$$\overline{\overline{x}} = x \tag{2.11}$$

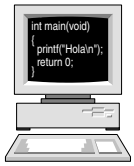
Ley de absorción

$$x + x \cdot y = x \qquad x \cdot (x + y) = x \tag{2.12}$$

Ambas propiedades pueden demostrarse fácilmente mediante manipulación algebraica aplicando los teoremas anteriores:

$$x + x \cdot y = x\,(1 + y) = x \cdot 1 = x \tag{2.13}$$

$$x \cdot (x + y) = x \cdot x + x \cdot y = x + x \cdot y = x \tag{2.14}$$



Realice el ejercicio 1

Propiedad asociativa

$$x \cdot (y \cdot z) = (x \cdot y) \cdot z \qquad x + (y + z) = (x + y) + z \tag{2.15}$$

Teorema del consenso

$$x \cdot y + \overline{x} \cdot z + y \cdot z = x \cdot y + \overline{x} \cdot z \qquad (x + y) \cdot (\overline{x} + z) \cdot (y + z) = (x + y) \cdot (\overline{x} + z) \tag{2.16}$$

Teorema de De Morgan

Este teorema recibe su nombre en honor a su descubridor: Augustus De Morgan.

$$\overline{(x + y)} = \overline{x} \cdot \overline{y} \qquad \overline{(x \cdot y)} = \overline{x} + \overline{y} \tag{2.17}$$

Este teorema se puede aplicar a expresiones con más términos:

$$\overline{(x + y + z)} = \overline{x} \cdot \overline{y} \cdot \overline{z} \qquad \overline{(x \cdot y \cdot z)} = \overline{x} + \overline{y} + \overline{z} \tag{2.18}$$

Incluso el teorema se puede generalizar para cualquier expresión booleana:

$$\overline{f\,(x_i, +, \cdot)} = f\,(\overline{x_i}, \cdot, +) \tag{2.19}$$

Es decir, el complemento de una expresión booleana f es la misma expresión pero cambiando todas las variables por sus complementos, las sumas por productos y los productos por sumas. Veamos un ejemplo:

$$\overline{x \cdot (y + z)} = \overline{x} + (\overline{y} \cdot \overline{z}) \tag{2.20}$$

La única precaución que hay que tomar al aplicar el teorema generalizado es no olvidarse de los paréntesis implícitos en los productos. Por ejemplo, si tenemos la expresión:

$$\overline{(x \cdot y + z)} \tag{2.21}$$

es fácil aplicar mal el teorema de De Morgan generalizado y obtener:

$$\overline{(x \cdot y + z)} = \overline{x} + \overline{y} \cdot \overline{z} \quad \text{ERROR} \tag{2.22}$$

en lugar de:

$$\overline{(x \cdot y + z)} = (\overline{x} + \overline{y}) \cdot \overline{z} \quad \text{Correcto} \tag{2.23}$$

Realice el ejercicio 2

Teorema expansión de Shannon

Para cualquier función booleana se cumple que:

$$f(x_1, \cdots, x_i, \cdots, x_n) = x_i \cdot f(x_1, \cdots, 1, \cdots, x_n) + \overline{x_i} \cdot f(x_1, \cdots, 0, \cdots, x_n) \tag{2.24}$$

$$f(x_1, \cdots, x_i, \cdots, x_n) = (x_i + \cdot f(x_1, \cdots, 0, \cdots, x_n)) \cdot (\overline{x_i} + f(x_1, \cdots, 1, \cdots, x_n)) \tag{2.25}$$

2.2. Funciones lógicas no básicas

Aunque cualquier función lógica puede expresarse a partir de las tres funciones básicas definidas por el álgebra de Boole (AND, OR y NOT), existen cuatro funciones lógicas que al ser usadas frecuentemente en la práctica se les ha dado un nombre y al igual que con las funciones básicas existen en el mercado circuitos que las implantan. En las siguientes secciones se muestran estas cuatro funciones.

2.2.1. Funciones NAND y NOR

Estas funciones no son más que una AND y una OR con la salida negada, es decir, la función NAND es:

$$f = \overline{a \cdot b}$$

y la función NOR es:

$$f = \overline{a + b}$$

ON	P	V	Al
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

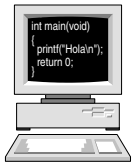
Cuadro 2.1: Tabla de verdad generada a partir de las especificaciones de la alarma.

2.2.2. Funciones XOR y XNOR

La función XOR aparece en los circuitos sumadores, como veremos más adelante. Dicha función se define como:

$$f = a \oplus b = \overline{a} \cdot b + a \cdot \overline{b}$$

Además esta función nos permite saber si dos variables lógicas son distintas, lo cual puede demostrarse fácilmente obteniendo la tabla de verdad a partir de la definición anterior.



La función XNOR es una XOR con la salida negada y nos permite averiguar si dos variables lógicas son iguales. Su definición es:

$$f = \overline{a \oplus b} = \overline{a} \cdot \overline{b} + a \cdot b$$

Realice los ejercicios 3 y 4

2.3. Formas normales de una función booleana

En general, el punto de partida en el proceso de diseño de un circuito lógico es la especificación del problema. Por ejemplo, supongamos que queremos diseñar un sistema de alarma para una casa con las siguientes especificaciones: *El sistema dispondrá de un interruptor para activar el sistema (ON), un sensor que detecta la apertura de la puerta (P) y otro sensor que detecta la apertura de la ventana (V). El circuito dispondrá de una salida para activar una bocina (Al). Dicha bocina ha de activarse cuando se abra una puerta o una ventana estando activado el sistema.* Como el lenguaje natural puede ser ambiguo, estas especificaciones se convierten a una tabla de verdad en la que se tienen en cuenta todas las posibles combinaciones de las variables de entrada, especificando el valor de las salidas del sistema para cada una de estas combinaciones. La tabla de verdad generada a partir de las especificaciones de la alarma se muestran en el cuadro 2.1.

El problema ahora consiste en convertir esta tabla de verdad en un circuito lógico a base de puertas. Para ello lo primero que debemos hacer es obtener una

expresión algebraica a partir de la tabla de verdad. No obstante, antes de hacer semejante hazaña debemos de realizar unas cuantas...

Definiciones

- **Literal:** llamaremos literal a una variable o a una variable negada. Por ejemplo son literales: $X, \overline{Y}, Al$.
- **Término de producto:** es un producto lógico de literales. Por ejemplo son términos de producto: $X \cdot Y \cdot Z, ON \cdot \overline{P}$.
- **Término de suma:** es una suma lógica de literales. Por ejemplo son términos de suma: $X + \overline{Y} + Z, ON + V$.
- **Minitérmino:** es un término de producto que contiene todos los literales de una función lógica. Así, en el ejemplo de la alarma son minitérminos las expresiones: $ON \cdot P \cdot V$ y $ON \cdot \overline{P} \cdot V$. Nótese que un minitérmino sólo vale 1 para una combinación de las entradas. Así el minitérmino $ON \cdot \overline{P} \cdot V$ sólo vale 1 si $ON = 1, P = 0$ y $V = 1$.
- **Maxitérmino:** es un término de suma que contiene todos los literales de una función lógica. Así, en el ejemplo de la alarma son maxitérminos las expresiones: $ON + P + V$ y $ON + P + \overline{V}$. Nótese que un maxitérmino sólo vale 0 para una combinación de las entradas. Así el maxitérmino $ON + P + \overline{V}$ sólo vale 0 si $ON = 0, P = 0$ y $V = 1$.

2.3.1. Obtención de una expresión lógica a partir de una tabla de verdad

Teniendo en cuenta que cada minitérmino sólo vale 1 para una combinación de las entradas, podemos asociar a cada fila de una tabla de verdad un único minitérmino que valdrá 1 sólo cuando las entradas tomen el valor de esa fila. De la misma manera, se puede asociar también a cada fila de una tabla de verdad el maxitérmino que vale cero para dicha fila. En el cuadro 2.2 se muestra la asociación de minitérminos y maxitérminos a la tabla del ejemplo de la alarma. Nótese que el minitérmino asociado a una fila puede obtenerse multiplicando todas las variables de la función, negando las que valen 0 en la fila. De la misma forma, el maxitérmino se obtiene sumando todas las variables y negando las que valen 1 en esa fila de la tabla.

Forma normal disyuntiva: suma de minitérminos

Si se suman todos los minitérminos asociados a las filas en las que la función lógica vale 1, tendremos una ecuación que representa el comportamiento lógico especificado en la tabla de verdad. A dicha expresión se le conoce como forma normal disyuntiva o suma de minitérminos. En el ejemplo de la alarma, la suma de minitérminos es la siguiente:

$$Al = ON \cdot \overline{P} \cdot V + ON \cdot P \cdot \overline{V} + ON \cdot P \cdot V$$

ON	P	V	Al	Minitérminos	Maxitérminos
0	0	0	0	$\overline{ON} \cdot \overline{P} \cdot \overline{V}$	$ON + P + V$
0	0	1	0	$\overline{ON} \cdot \overline{P} \cdot V$	$ON + P + \overline{V}$
0	1	0	0	$\overline{ON} \cdot P \cdot \overline{V}$	$ON + \overline{P} + V$
0	1	1	0	$\overline{ON} \cdot P \cdot V$	$ON + \overline{P} + \overline{V}$
1	0	0	0	$ON \cdot \overline{P} \cdot \overline{V}$	$\overline{ON} + P + V$
1	0	1	1	$ON \cdot \overline{P} \cdot V$	$\overline{ON} + P + \overline{V}$
1	1	0	1	$ON \cdot P \cdot \overline{V}$	$\overline{ON} + \overline{P} + V$
1	1	1	1	$ON \cdot P \cdot V$	$\overline{ON} + \overline{P} + \overline{V}$

Cuadro 2.2: Asociación de maxitérminos y minitérminos a la tabla de verdad..

En ciertas ocasiones es conveniente usar una notación más simple para expresar una suma de minitérminos. Teniendo en cuenta que cada minitérmino está asociado a una fila de la tabla de verdad, basta con enumerar los números de las filas de dicha tabla en las que la función lógica vale 1. Así la suma de minitérminos anterior puede expresarse como:

$$\sum_{ON,P,V} (5, 6, 7)$$

Forma normal conjuntiva: producto de maxitérminos

Si se multiplican todos los minitérminos asociados a las filas de una tabla de verdad en las que la función lógica vale 0 también tenemos una ecuación que representa la función lógica especificada en dicha tabla de verdad. A esta expresión se le conoce como forma normal conjuntiva o producto de maxitérminos. En el ejemplo de la alarma la forma normal conjuntiva es:

$$AL = (ON + P + V) \cdot (ON + P + \overline{V}) \cdot (ON + \overline{P} + V) \cdot (ON + \overline{P} + \overline{V}) \cdot (\overline{ON} + P + V)$$

Al igual que con la suma de minitérminos, también existe una notación compacta para expresar el producto de maxitérminos:

$$\prod_{ON,P,V} (0, 1, 2, 3, 4)$$

Tanto la forma normal conjuntiva como la forma normal disyuntiva nos permiten obtener una ecuación lógica a partir de una tabla de verdad. A partir de este momento tenemos toda la potencia del álgebra de Boole para manipular dicha expresión y simplificarla o también podemos implantarla usando puertas lógicas.

También conviene destacar que, como se acaba de demostrar, cualquier tabla de verdad se puede expresar como una suma de productos o como un producto de sumas. Por tanto, tal como se había adelantado en la introducción, cualquier circuito digital puede implantarse mediante puertas AND, OR y NOT. Por otro lado, esta propiedad es la que permite diseñar circuitos lógicos programables como

el mostrado en la figura 1.9 formado por dos niveles de puertas AND y OR para implantar funciones lógicas expresadas como suma de productos.

2.3.2. *Otras formas de representación: simplificación*

Las ecuaciones obtenidas en la sección anterior implementan la tabla de verdad de partida y por tanto son perfectamente válidas. No obstante, se pueden aplicar los teoremas del álgebra de Boole para simplificar la ecuación y poder así implantarla usando menos puertas (o puertas de menos entradas). Si partimos de la suma de minitérminos, aplicando el teorema de idempotencia (ecuación 2.10) podemos duplicar el último minitérmino:

$$Al = ON \cdot \overline{P} \cdot V + ON \cdot P \cdot \overline{V} + ON \cdot P \cdot V = ON \cdot \overline{P} \cdot V + ON \cdot P \cdot \overline{V} + ON \cdot P \cdot V + ON \cdot P \cdot V$$

Si nos fijamos en el primer término y el tercero, podemos sacar factor común a $ON \cdot V$. De la misma forma, podemos sacar factor común a $ON \cdot P$ en los términos segundo y cuarto, obteniendo así la expresión:

$$Al = ON \cdot V \cdot (\overline{P} + P) + ON \cdot P \cdot (\overline{V} + V)$$

Aplicando ahora los teoremas del elemento complementario (ecuación 2.9) y del elemento identidad (ecuación 2.5), obtenemos:

$$Al = ON \cdot V + ON \cdot P \tag{2.26}$$

Esta ecuación ya no es una forma normal, pero sigue siendo una suma de productos y se le denomina precisamente así. La diferencia con la forma normal disyuntiva es obvia: es mucho más simple de implementar, al necesitar sólo dos puertas AND y una OR, ambas de dos entradas; frente a tres puertas AND y una OR, las cuatro de tres entradas.

El mismo procedimiento puede realizarse con la forma normal conjuntiva. En este caso se aplica el teorema de idempotencia para duplicar el primer término:

$$\begin{aligned} AL &= (ON + P + V) \cdot (ON + P + \overline{V}) \cdot (ON + \overline{P} + V) \cdot (ON + \overline{P} + \overline{V}) \cdot (\overline{ON} + P + V) \\ &= (ON + P + V) \cdot (ON + P + V) \cdot (ON + P + \overline{V}) \cdot (ON + \overline{P} + V) \cdot (ON + \overline{P} + \overline{V}) \\ &\quad \cdot (\overline{ON} + P + V) \end{aligned}$$

Aplicando la propiedad distributiva respecto de la suma podemos sacar factor común a $ON + P$ entre los términos 2 y 3, a $P + V$ entre los términos 1 y 6, a $ON + \overline{P}$ entre los términos 4 y 5. Se obtiene entonces:

$$AL = (ON + P + V \cdot \overline{V}) \cdot (ON \cdot \overline{ON} + P + V) \cdot (ON + \overline{P} + V \cdot \overline{V})$$

Que aplicando los teoremas del elemento complementario (ecuación 2.9) y del elemento identidad (ecuación 2.5), se convierte en:

$$AL = (ON + P) \cdot (P + V) \cdot (ON + \overline{P}) = (ON + P) \cdot (ON + \overline{P}) \cdot (P + V)$$

En esta expresión se puede volver a aplicar la propiedad distributiva respecto de la suma para sacar factor común a la variable *ON*, obteniendo:

$$AL = (ON + P \cdot \overline{P}) \cdot (P + V)$$

Que, después de tener en cuenta que $P \cdot \overline{P} = 0$, se transforma en:

$$AL = ON \cdot (P + V)$$

Nuevamente la expresión obtenida, aunque no es una forma normal, es un producto de sumas,[1] pero mucho más simple que la forma normal conjuntiva.

2.4. Simplificación usando diagramas de Karnaugh

El método de simplificación mostrado en la sección anterior se basa en buscar dos términos de producto, o dos términos de suma, en los que una variable aparece negada y sin negar; para a continuación sacar factor común y simplificar dicha variable. El problema es que no siempre es fácil encontrar estos términos. Los diagramas de Karnaugh nos ayudan precisamente a eso, aunque tienen como limitación el no ser aplicables a funciones de más de cinco variables.[2]

2.4.1. Diagramas de Karnaugh para funciones de 2, 3, 4 y 5 variables

En primer lugar hay que tener en cuenta que un diagrama de Karnaugh no es más que una representación gráfica de una tabla de verdad en la que cada casilla representa una fila de la tabla de verdad. La particularidad de estos diagramas es que las casillas adyacentes representan filas de la tabla de verdad entre las que difiere sólo el valor de una de las entradas.

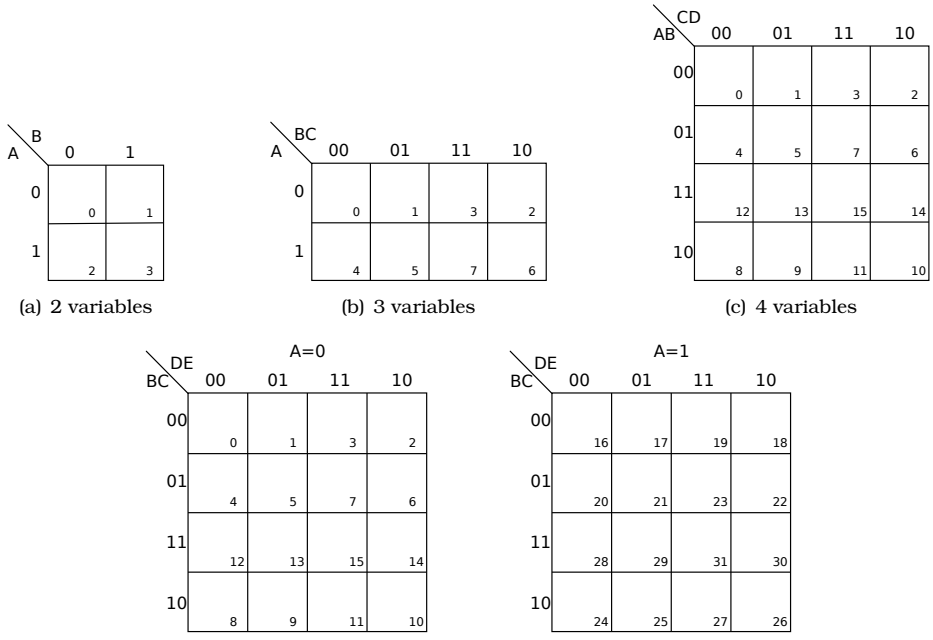
Existen diagramas de 2, 3, 4 y 5 variables, las cuales se muestran en la figura 2.1. En dicha figura se muestra para cada casilla el número de la fila de la tabla de verdad asociada a dicha casilla, el cual se corresponde con el número en binario formado por las variables del diagrama.

2.4.2. Obtención del diagrama de Karnaugh a partir de una tabla de verdad

Para obtener el diagrama de Karnaugh de una función lógica se parte de la tabla de verdad y se escribe el valor de cada fila de la tabla en su casilla correspondiente del diagrama de Karnaugh. En la figura 2.2 se muestra el diagrama de Karnaugh obtenido para el ejemplo de la alarma. Al mismo resultado se llega si en lugar de

[1] Se puede definir un término de suma como la suma de uno o más literales. *ON* es obviamente un término de suma de un literal.

[2] Existen algoritmos de simplificación más complejos que permiten simplificar funciones de más variables, como por ejemplo el algoritmo de Quine-McCluskey. No obstante el estudio de estos algoritmos se salen del alcance de este texto.



(a) 2 variables (b) 3 variables (c) 4 variables

(d) 5 variables

Figura 2.1: Mapas de Karnaugh.

partir de la tabla de verdad partimos de la función lógica en forma de suma de productos, que para el ejemplo de la alarma es:

$$\sum_{ON,P,V} (5, 6, 7)$$

Obtención de la función lógica simplificada a partir del diagrama de Karnaugh

Como se ha dicho anteriormente, dos casillas adyacentes del diagrama de Karnaugh representan filas de la tabla de verdad en las que una variable toma dos valores distintos y el resto tienen el mismo valor; por lo que el diagrama nos permite identificar términos que se pueden simplificar.

Para ilustrar el proceso usaremos el ejemplo de la alarma. El primer paso es decidir si queremos obtener como ecuación final una suma de productos o un producto de sumas. En el primer caso tenemos que buscar casillas que estén a 1 y que sean adyacentes a alguna otra que esté también a 1. Si nos fijamos en la figura 2.2 tenemos como adyacentes las casillas nº 5 y 7 y las nº 7 y 6, las cuales se han marcado en el diagrama. Las casillas 5 y 7 representan los minitérminos $ON \cdot \overline{P} \cdot V$ y $ON \cdot P \cdot V$ respectivamente. La suma de ambos minitérminos se puede simplificar

(a) Tabla de verdad

Nº Fila	ON	P	V	Al
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	0
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

(b) Diagrama de Karnaugh

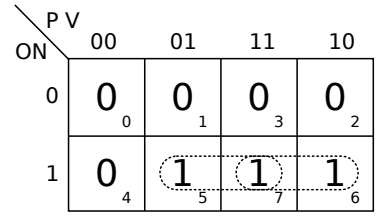


Figura 2.2: Obtención del diagrama de Karnaugh para una función.

por tanto a $ON \cdot V$. De la misma forma, las casillas 7 y 6 representan $ON \cdot P \cdot V$ y $ON \cdot P \cdot \overline{V}$ respectivamente y su suma se simplifica a $ON \cdot P$. Como para obtener la ecuación lógica es necesario sumar los minitérminos en los que la función vale 1, esta suma será igual en este caso a:

$$Al = ON \cdot V + ON \cdot P \tag{2.27}$$

Que como puede comprobar es el mismo que el obtenido en la sección 2.3.2 (véase la ecuación 2.26). Además, si se fija con detenimiento observará que el proceso seguido ha sido el mismo, sólo que usando el diagrama de Karnaugh ha sido fácil identificar las variables que se pueden simplificar y también ver que el minitérmino nº 7 es necesario duplicarlo para la simplificación.

Por último, otra buena noticia es que no es necesario realizar el proceso que se acaba de describir para obtener el término de producto resultante de un grupo de casillas. Al contrario, éste se forma multiplicando las variables que no cambian dentro del grupo, negando las que valen 0. Así, en el grupo formado por las casillas 5 y 7 las variables que no cambian dentro del grupo son ON y V. Como ambas valen 1, el grupo genera el término de producto $ON \cdot V$. De la misma forma el grupo formado por las casillas 7 y 6 genera el término de producto $ON \cdot P$.

Por otro lado, la adyacencia se generaliza a grupos de 2^n casillas que formen un rectángulo. Así, el método general puede resumirse en el siguiente algoritmo:

1. Formar grupos con las casillas que estén a uno hasta cubrir todos los unos. Para ello se comienza buscando las casillas aisladas, a continuación los grupos de 2 casillas que no pueden ampliarse a 4, luego los grupos de 4 que no pueden ampliarse a 8; y así sucesivamente. Nótese que los grupos han de ser cuadrados o rectángulos de 2, 4, 8 o 16 casillas.

2. Obtener los términos de producto generados por cada casilla multiplicando las variables que no cambian dentro del grupo, negando las que valgan 0.

3. Sumar todos los términos obtenidos en el proceso anterior.

Ejemplos

Lo mejor para ilustrar el proceso es ver algunos ejemplos. En primer lugar simplificaremos la ecuación dada por la suma de productos:

$$F = \sum_{A,B,C,D} (1, 3, 4, 5, 6, 7, 11, 12, 14, 15)$$

En la figura 2.3(a) se muestra el diagrama resultante al agrupar todos los unos de la función. El grupo formado por las casillas 1, 3, 5 y 7 forma el término de producto $\overline{A} \cdot D$, el grupo 3, 7, 11, y 15 genera $C \cdot D$ y por último el grupo formado por 4, 6, 12 y 14 genera $B \cdot \overline{D}$. Nótese que en este último caso se ha tenido en cuenta que las casillas que están en los bordes del diagrama son adyacentes a las del borde contrario. Así las casillas 4 y 6 y las 12 y 14 son adyacentes, permitiendo formar un grupo de 4 casillas. Sumando los términos generados por cada grupo obtenemos la ecuación simplificada en forma de suma de productos:

$$F = \overline{A} \cdot D + C \cdot D + B \cdot \overline{D}$$

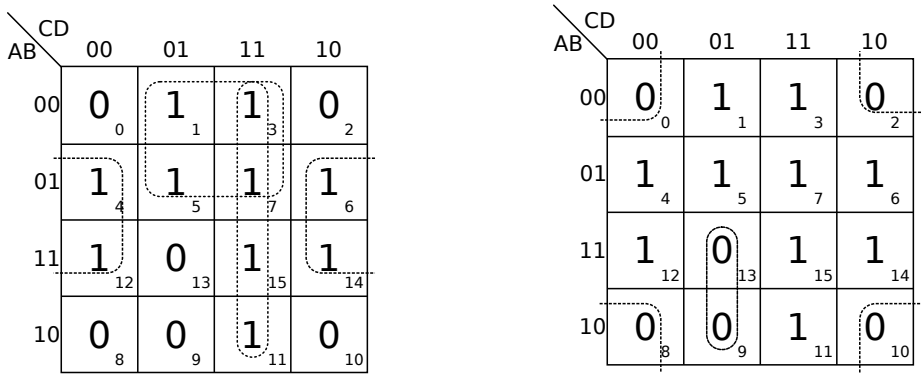
El mismo proceso se puede realizar usando los ceros del diagrama. En este caso cada grupo de ceros genera un término de suma, el cual se obtiene sumando las variables que no cambian en todo el grupo negando las que valen 1. Así, según podemos observar en la figura 2.3(b), podemos formar dos grupos de ceros: el formado por las casillas 1, 2, 8 y 10, que genera el término de suma $B + D$ y el formado por las casillas 9 y 13, que genera el término de suma $\overline{A} + C + \overline{D}$. Multiplicando ambos términos se obtiene la ecuación simplificada en forma de producto de sumas:

$$F = (B + D) \cdot (\overline{A} + C + \overline{D})$$

2.4.3. Simplificación con condiciones libres

Existen ocasiones en las que ciertas combinaciones de las entradas no se dan en la práctica. Por ejemplo, en el circuito de la figura 2.4 sólo son válidas las entradas 1110, 1101, 1011 y 0111. El resto de combinaciones es imposible que se den salvo que se estropee el circuito. En estos casos se puede aprovechar este hecho para simplificar más el circuito.

Supongamos que queremos diseñar el circuito de la figura 2.4 para que active su salida cuando el conmutador esté en la posición a o b. La tabla de verdad quedaría tal como se muestra en la figura 2.5, en donde en la salida correspondiente a las líneas de la tabla de verdad cuya combinación de entradas no pueden darse nunca se ha puesto una X. Esta X, denominada condición libre (*don't care* en inglés), significa que nos da igual el valor que tome la tabla de verdad en esa fila, ya que nunca se va a dar en la práctica. En la figura 2.5(b) se muestra el diagrama de Karnaugh correspondiente a dicha tabla. Para obtener las ecuaciones a partir del diagrama usamos las X como si fuesen "comodines". Puesto que nos da igual el valor que tome la tabla de verdad en las casillas que están a X, podemos asignar un 1 a



(a) Agrupación de unos

(b) Agrupación de ceros

Figura 2.3: Mapas de Karnaugh de $\sum_{A,B,C,D}(1,3,4,5,6,7,11,12,14,15)$

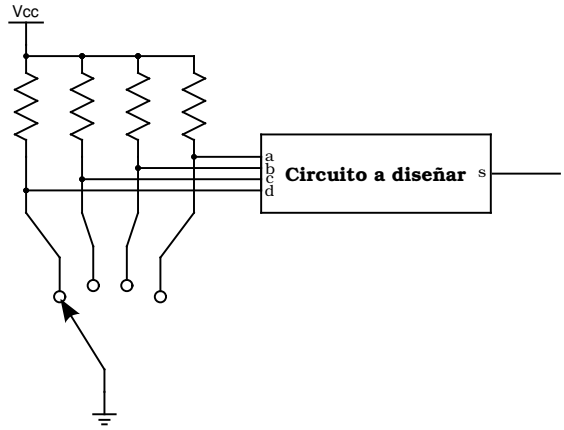


Figura 2.4: Circuito con combinaciones de entradas imposibles

(a) Tabla de verdad

a	b	c	d	S
0	0	0	0	X
0	0	0	1	X
0	0	1	0	X
0	0	1	1	X
0	1	0	0	X
0	1	0	1	X
0	1	1	0	X
0	1	1	1	1
1	0	0	0	X
1	0	0	1	X
1	0	1	0	X
1	0	1	1	1
1	1	0	0	X
1	1	0	1	0
1	1	1	0	0
1	1	1	1	X

(b) Diagrama de Karnaugh

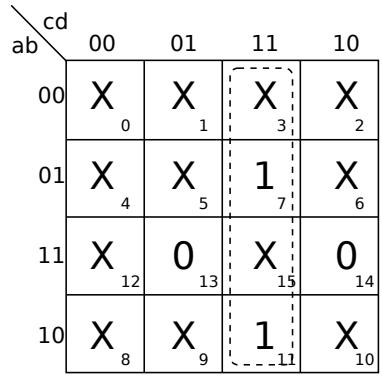


Figura 2.5: Tabla de verdad y diagrama de Karnaugh del circuito de la figura 2.4.

las casillas 3 y 15 y un 0 al resto. Con ello, se puede formar el grupo mostrado en la figura, con lo que la ecuación del circuito queda igual a:

$$S = c \cdot d$$

Si por el contrario no se hubiesen tenido en cuenta las condiciones libres, todas las casillas estarían a 0, con lo que no se podría realizar ninguna simplificación, quedando la ecuación lógica de la salida de la forma:

$$S = \overline{a} \cdot b \cdot c \cdot d + a \cdot \overline{b} \cdot c \cdot d$$

Que como puede observarse es bastante más compleja que la ecuación anterior.

2.5. Ejercicios

1. Simplifique las siguientes expresiones, indicando qué teoremas ha aplicado para ello:

$$F_1 = W \cdot X \cdot Y \cdot Z \cdot \left(W \cdot X \cdot Y \cdot \overline{Z} + W \cdot \overline{X} \cdot Y \cdot Z + \overline{W} \cdot X \cdot Y \cdot Z + W \cdot X \cdot \overline{Y} \cdot Z\right)$$

$$F_2 = A \cdot B + A \cdot B \cdot \overline{C} \cdot D + A \cdot B \cdot D \cdot \overline{E} + A \cdot B \cdot \overline{C} \cdot E + \overline{C} \cdot D \cdot E$$

2. Simplifique las siguientes expresiones lógicas usando el teorema de Morgan generalizado:

$$F_3 = \overline{\left[A + B \cdot \left(C + \overline{A}\right)\right]}$$
$$F_4 = \overline{\left[\overline{A} + \overline{B} \cdot (C + A)\right]}$$

3. Obtenga las tablas de verdad de las funciones NAND, NOR, XOR y XNOR a partir de sus definiciones mostradas en la sección 2.2.
4. Demuestre que las funciones NAND y NOR no cumplen la propiedad asociativa, es decir:

$$\overline{\overline{X \cdot Y} \cdot Z} \neq \overline{X \cdot \overline{Y \cdot Z}}$$
$$\overline{\overline{X + Y} + Z} \neq \overline{X + \overline{Y + Z}}$$

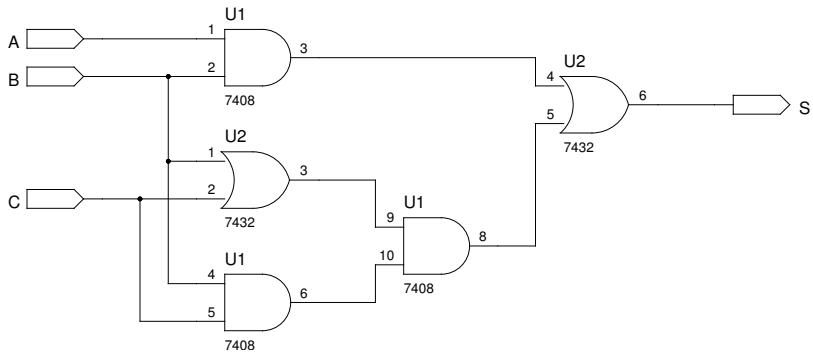
5. Simplifique usando diagramas de Karnaugh las siguientes funciones lógicas expresadas como suma de minitérminos o producto de maxitérminos. Para cada una de las funciones lógicas ha de obtener la expresión más simple posible, ya sea ésta una suma de productos o un producto de sumas. Además deberá dibujar un circuito digital para implantar cada una de las funciones lógicas.

$$F_1 = \sum_{A,B,C,D} (0, 1, 2, 3, 8, 9, 10, 11) \tag{2.28}$$
$$F_2 = \prod_{A,B,C,D} (0, 4, 10, 11, 14, 15) \tag{2.29}$$
$$F_3 = \sum_{A,B,C,D} (0, 2, 8, 10) \tag{2.30}$$
$$F_4 = \sum_{A,B,C,D,E} (0, 2, 5, 7, 8, 10, 13, 15, 21, 23) \tag{2.31}$$

6. En una central solar se dispone de 4 grupos de paneles y se desea monitorizar su funcionamiento. Para ello cada grupo dispone de un sensor que se activa (1) si el grupo está funcionando correctamente y se desactiva (0) en caso de que se detecte un fallo en el grupo. Diseñe un circuito que a partir de la información proporcionada por estos sensores active una señal cuando falle sólo uno de los grupos, otra cuando fallen dos y otra cuando fallen tres o más grupos.
7. En su casa se va a instalar un sistema de aire acondicionado por zonas. Este sistema dispone de una máquina central y de una serie de conductos que transportan el aire a cada una de las cuatro habitaciones de su casa. Cada habitación dispone de una rejilla motorizada que permite abrir o cerrar el paso del aire, para poder así regular su climatización. La máquina dispone de dos ventiladores, uno de bajo caudal y otro de alto caudal. El funcionamiento de la instalación ha de ser tal que cuando estén abiertas una o dos rejillas

funcione sólo el ventilador de bajo caudal y cuando estén abiertas tres o más rejillas funcione sólo el ventilador de alto caudal. Diseñe un circuito que disponiendo como entradas el estado de cada una de las rejillas (1 abierta, 0 cerrada) controle el encendido y apagado de cada uno de los dos ventiladores (1 encendido, 0 apagado).

8. Un amigo suyo le ha proporcionado el siguiente circuito. Como no se fía mucho de él, compruebe si se puede obtener un circuito que implante la misma función con menos puertas. Para ello obtenga la ecuación lógica del circuito y simplifíquela usando las propiedades del álgebra de Boole.



CAPÍTULO 3

Sistemas de numeración

En este capítulo se estudian los sistemas de numeración usados en sistemas digitales, las técnicas para convertir entre bases de numeración y cómo hacer operaciones básicas en binario.

3.1. Introducción

A lo largo de la historia se han usado diferentes sistemas de numeración. Uno que se sigue usando en la actualidad, aunque muy poco, es el sistema romano. Como sabrá, en estos números existen una serie de símbolos con un valor fijo y unas reglas muy sencillas para interpretarlos. Básicamente hay que sumar todos los símbolos, salvo cuando uno de menor valor precede a uno de mayor valor, en cuyo caso se resta el menor del mayor. Por ejemplo:

$$\text{XXVII} = 10 + 10 + 5 + 1 + 1 = 27$$
$$\text{IX} = 10 - 1 = 9$$

El mayor inconveniente de este sistema de numeración, aparte de su legibilidad, es que no es fácil realizar operaciones con ellos. Existen algoritmos para sumar y restar que son relativamente fáciles. Sin embargo la multiplicación es compleja y la división es imposible. Aunque en [Chu-Carroll, 2006] se muestran los algoritmos para operar con números romanos, ciertamente es más cómodo realizar la operación convirtiendo los números a decimal, hacer la operación y luego volver a convertir a números romanos:

$$\text{XXVII} + \text{IX} = 27 + 9 = 36 = \text{XXXVI}$$

Debido a esto, se estableció un sistema de numeración más potente, el decimal, que aparte de ser mucho más legible y poder representar números grandes,[1] permite realizar operaciones fácilmente.

[1] El mayor símbolo de los números romanos es el M (1000), por lo que para representar por ejemplo el número 1.000.000 sería necesario escribir 1.000 M seguidas.

3.2. Sistemas de numeración posicionales

El sistema de numeración decimal es mucho más útil debido a que es un sistema de numeración posicional, es decir, el valor de cada dígito depende de la posición en la que se encuentra dentro del número. Así, en el número 27 el 2 tiene un valor de 2 decenas, mientras que en el número 72, el mismo 2 tiene un valor de 2 unidades. Por el contrario, en los números romanos cada letra tiene siempre el mismo valor, independientemente de donde se encuentre. Así, en el número XI la X vale 10, y en el IX, la X también vale 10.

En concreto, en un sistema de numeración posicional cada cantidad se representa por una secuencia de símbolos (dígitos), de forma que el valor representado es igual a la suma de los productos de cada símbolo por un valor (peso) que depende de la posición del símbolo.

3.2.1. Representación del tiempo

El tiempo se puede representar mediante un sistema posicional. Por ejemplo, 1:33:27 equivale a 1 hora, 33 minutos, 27 segundos, o lo que es lo mismo: $1 \times 60 \times 60 + 33 \times 60 + 27$ segundos.

Si el tiempo a representar es mayor que un día, se añadiría un dígito más por la izquierda: 2:1:33:27 equivale a 2 días, 1 hora, 33 minutos, 27 segundos, o lo que es lo mismo: $2 \times 24 \times 60 \times 60 + 1 \times 60 \times 60 + 33 \times 60 + 27$ segundos.

El usar 60 como peso para horas, minutos y segundos es una herencia de los Sumerios, que empezaron a usarlo en el año 2000 AC. La razón de usar 60 es que es divisible por 12 números: 1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 30 y 60. Esto hace que muchas fracciones de los números en base 60 (sexagesimales) sean enteras. En la actualidad los números sexagesimales se usan para expresar tiempo, ángulos y coordenadas geográficas.

3.2.2. Sistemas de numeración polinómicos

Cuando en un sistema de numeración posicional los pesos se incrementan según una progresión geométrica. decimos que es un sistema de numeración polinómico. En estos casos la progresión geométrica se forma con las potencias de un número al que se denomina base.

El sistema decimal es el ejemplo más conocido de este tipo de sistema de numeración. Así cuando escribimos 1327, la cantidad que se está representando es:

$$1327 = 1 \times 1000 + 3 \times 100 + 2 \times 10 + 7 \times 1 = 1 \times 10^3 + 3 \times 10^2 + 2 \times 10^1 + 7 \times 10^0$$

Como se puede observar, la base usada en el sistema decimal es 10 (de ahí su nombre).

En general, dado un número $d_{n-1}d_{n-2}\cdots d_1 d_0$, en una base b, la cantidad rep-

resentada por el número es:

$$d_{n-1}d_{n-2}\cdots d_1 d_0 = d_{n-1}\times b^{n-1} + d_{n-2}\times b^{n-2} + \cdots + d_1\times b^1 + d_0\times b^0 = \sum_{i=0}^{n-1} d_i \times b^i$$

Si el número contiene una coma, la fórmula anterior se generaliza a:

$$d_{n-1}\cdots d_1 d_0, d_{-1}\cdots d_{-m} = d_{n-1}\times b^{n-1} + \cdots + d_1\times b^1 + d_0\times b^0 + \\ + d_{-1}\times b^{-1} + \cdots + d_{-m}\times b^{-m} = \sum_{i=-m}^{n-1} d_i \times b^i$$

Por ejemplo, el número 123,42 equivale a:

$$123,42 = 1\times 10^2 + 2\times 10^1 + 3\times 10^0 + 4\times 10^{-1} + 2\times 10^{-2} = \\ = 1\times 100 + 2\times 10 + 3\times 1 + 4\times 0,1 + 2\times 0,01$$

Por otro lado, es conveniente resaltar que el número de símbolos necesarios para representar cada dígito es igual a la base. Así en el sistema decimal se necesitan diez símbolos: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9.

3.2.3. El sistema binario

Nosotros usamos el sistema decimal porque tenemos 10 dedos.[2] Sin embargo, en un sistema digital sólo existen dos símbolos: 0 y 1. Por tanto para representar un número sólo puede usarse base 2. No obstante, el manejo de números en binario es exactamente igual al de números decimales. Por ejemplo el número 1001_2 representa[3] la cantidad:

$$1001_2 = 1\times 2^3 + 0\times 2^2 + 0\times 2^1 + 1\times 2^0 = 1\times 8 + 0\times 4 + 0\times 2 + 1\times 1 = 9$$

A pesar de esto, el manejo de números binarios es un poco incómodo por dos razones:

- Para representar una cantidad se necesita una gran cantidad de dígitos, lo cual hace que sean difíciles de memorizar, escribir, etc. Se dice que estos números son "poco expresivos".
- La falta de costumbre hace que nos sea difícil hacernos una idea de la cantidad que representa un número en binario, cosa que no ocurre en decimal.

No obstante no todo son inconvenientes. Los números binarios también tienen sus ventajas.

- Al necesitar sólo dos símbolos pueden almacenarse y procesarse en un sistema digital.

[2] Los mayas usaban un sistema de numeración en base 20 [Wikipedia, 2010]. Es de suponer que usaban los dedos de las manos y de los pies para contar.

[3] Para evitar confusiones, en este texto los números que estén en una base distinta a la decimal se distinguirán poniendo su base como un subíndice del número.

- Las operaciones son muy sencillas. Por ejemplo las reglas de la suma son sólo 4 frente a las 100 de los números decimales:

$$0 + 0 = 0$$
$$0 + 1 = 1$$
$$1 + 0 = 1$$
$$1 + 1 = 10$$

  Nótese que en la última ecuación 10 representa el número dos en decimal, no el 10.[4]

En conclusión los sistemas digitales sólo pueden trabajar con números binarios, pero a nosotros nos resulta incómodo trabajar directamente con ellos. La solución consiste en dejar a los sistemas digitales que trabajen en binario, trabajar nosotros en decimal y realizar las conversiones de bases cuando tengamos que transferir un número a/desde un sistema digital.

3.3. Conversión entre bases

Como se acaba de decir, es frecuente en la práctica tener que convertir números de binario a decimal y viceversa. En esta sección se expone cómo realizar estas conversiones.

3.3.1. Conversión de binario a decimal

Para pasar de binario (o de cualquier base b) a decimal basta con aplicar la fórmula:

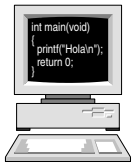
$$N = \sum_{i=0}^{n-1} d_i \times b^i$$

En donde b es la base, que recordemos que en binario es 2. Por ejemplo, para convertir el número 1011001_2 a decimal, basta con calcular:

$$1011001_2 = 1 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 89$$

Si el número a convertir tiene una coma binaria la conversión se realiza usando la fórmula generalizada:

$$N = \sum_{i=-m}^{n-1} d_i \times b^i$$



Realice el ejercicio 1

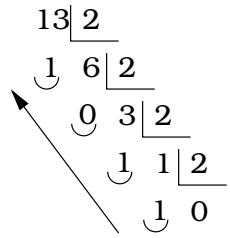
Por ejemplo, la conversión 111,011 a decimal se realiza:

$$111,011_2 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 + 0 \times 2^{-1} + 1 \times 2^{-2} + 1 \times 2^{-3} = 7,375$$

[4] Existe una frase graciosa, sólo apta para "frikis": *Existen 10 tipos de personas: las que saben binario y las que no.*

3.3.2. Conversión de decimal a binario

Para convertir un número decimal a binario existen varias técnicas, aunque la más usada es la de divisiones sucesivas. Esta técnica consiste en ir dividiendo el número a convertir entre dos e ir anotando los restos de las divisiones parciales. Se termina de dividir cuando el resultado es cero, formándose el número binario leyendo los restos obtenidos de derecha a izquierda. Lo mejor es ilustrar el proceso mediante un ejemplo. Así, la conversión del número 13 a binario se realiza:



Por tanto, $13 = 1101_2$.

El algoritmo anterior sólo es válido para la parte entera del número. La parte decimal se convierte mediante multiplicaciones sucesivas. Para ello se toma dicha parte decimal y se multiplica sucesivamente por dos. La parte entera del resultado es el bit resultante de la conversión y la parte decimal se vuelve a multiplicar por dos. El algoritmo finaliza cuando la parte decimal del resultado sea cero o cuando hayamos obtenido el número de bits deseado para la parte fraccionaria. El número en binario se forma leyendo de arriba a abajo las partes enteras obtenidas de las multiplicaciones. Al igual que antes, lo mejor es ilustrar el algoritmo mediante un ejemplo:

$$0,375 \times 2 = 0,75$$
$$0,75 \times 2 = 1,5$$
$$0,5 \times 2 = 1,0$$

Por tanto, $0,375 = 0,011_2$ y teniendo en cuenta la conversión anterior, $13,375 = 1101,011_2$.

Si la parte fraccionaria no es divisible entre dos no se puede representar con un número finito de bits. Por ejemplo, la conversión a binario de $0,3$ se realiza:

$$0,3 \times 2 = 0,6$$
$$0,6 \times 2 = 1,2$$
$$0,2 \times 2 = 0,4$$
$$0,4 \times 2 = 0,8$$
$$0,8 \times 2 = 1,6$$
$$0,6 \times 2 = 1,2$$

Como se puede observar, a partir de la sexta línea se repite el resultado obtenido de la multiplicación, por lo que el resultado nunca podrá tener cero como parte fraccionaria. Es decir, $0,3 = 0,0\widehat{1001}$. En estos casos, dado que en un sistema digital el número de bits es finito, se termina la conversión cuando se obtengan los bits reservados a la parte fraccionaria del número. En este caso la representación binaria no será exacta, cometiéndose un **error de cuantización**. En el ejemplo anterior, si tenemos que representar $0,3$ con sólo 6 bits para la parte fraccionaria, obtendremos:

$$0,010011_2 = 0 \times 2^{-1} + 1 \times 2^{-2} + 0 \times 2^{-3} + 0 \times 2^{-4} + 1 \times 2^{-5} + 1 \times 2^{-6} = 0,296875$$

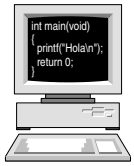
El error de cuantización cometido es igual a:

$$e = \frac{0,3 - 0,296875}{0,3} \cdot 100\,\% = 1,04\,\%$$

En general, el error de cuantización se obtiene mediante la fórmula:

$$e = \frac{numero_{original} - numero_{aproximado}}{numero_{original}} \cdot 100\,\%$$

Por ejemplo, al convertir $13,3$ a binario usando 6 bits para la parte fraccionaria, el resultado es $1101,010011_2$, siendo el error de cuantización:



Realice el ejercicio 2

$$e = \frac{13,3 - 13,296875}{13,3} \cdot 100\,\% = 0,0235\,\%$$

3.4. Rangos

En un sistema digital los números binarios han de representarse mediante un número fijo de bits. Por ejemplo, para transmitir un número binario dentro de un circuito normalmente se usa un conjunto de conductores de forma que cada uno transporta un bit. Obviamente una vez que se ha fabricado el circuito estos conductores no pueden ampliarse, por lo que todos los números que se transmitan usando dichos conductores han de tener el mismo número de bits.

La repercusión de tener que reservar de antemano un número de bits para representar los números binarios en un sistema digital es que el rango de valores que se pueden representar es limitado. Por ejemplo, si se dispone de tres bits, el

rango de números que es posible representar es:

$$000 \rightarrow 0$$
$$001 \rightarrow 1$$
$$010 \rightarrow 2$$
$$011 \rightarrow 3$$
$$100 \rightarrow 4$$
$$101 \rightarrow 5$$
$$110 \rightarrow 6$$
$$111 \rightarrow 7$$

En general, es fácil demostrar que el rango de valores que se pueden representar con n bits es $0 \leftrightarrow 2^n - 1$. Así por ejemplo, para tres bits el rango será $0 \leftrightarrow 2^3 - 1 \Leftrightarrow 0 \leftrightarrow 7$. Esto significa que no se podrá representar ningún número mayor que 7. Si se desea representar un número mayor será necesario usar más bits.

3.5. Sistemas hexadecimal y octal

Tal como se ha visto en la sección 3.3, la conversión entre binario y decimal es relativamente compleja. Por otro lado, trabajar con números binarios no es cómodo debido a la gran cantidad de dígitos necesarios. Debido a esto se usan en la práctica otras bases que permiten una fácil conversión a binario y, al necesitar menos dígitos para representar los números, hacen que su uso sea más amigable por parte del hombre. Estas bases son aquellas que sean múltiplos de dos, pues entonces se puede demostrar que la conversión puede realizarse dividiendo el número en grupos de bits y convirtiendo cada uno por separado, lo cual es más fácil que convertir el número completo como ocurre en decimal. Las dos bases usadas son la octal (base 8) y la hexadecimal (base 16).

En el cuadro 3.1 se muestra la equivalencia entre decimal, binario, octal y hexadecimal. Nótese que, tal como se dijo anteriormente, para la base 8 necesitamos sólo ocho símbolos, por lo que obviamente se han elegido los números de 0 a 7. Para representar los números en base 16 se necesita tener 16 símbolos. Como se puede apreciar en el cuadro, se han elegido como símbolos del 0 al 9 y para los seis restantes, en lugar de inventar garabatos nuevos se han usado las letras de la A a la F, asignándole los valores mostrados en el cuadro.

3.5.1. Conversión entre binario y octal

La conversión de binario a octal se realiza agrupando los bits de tres en tres empezando por el bit menos significativo[5]. Si es necesario se añadirán ceros por la

[5]El bit menos significativo es el que está a la derecha del número y el más significativo es el que está a la izquierda.

Decimal	Binario	Octal	Hexadecimal
0	00000	00	0
1	00001	01	1
2	00010	02	2
3	00011	03	3
4	00100	04	4
5	00101	05	5
6	00110	06	6
7	00111	07	7
8	01000	10	8
9	01001	11	9
10	01010	12	A
11	01011	13	B
12	01100	14	C
13	01101	15	D
14	01110	16	E
15	01111	17	F
16	10000	20	10

Cuadro 3.1: Equivalencia entre decimal, binario, octal y hexadecimal.

izquierda para completar el último grupo. Una vez agrupados los bits, se convierte cada grupo por separado. Lo mejor, como siempre, es realizar unos ejemplos:

$$100111101001_2 = 100\,111\,101\,001_2 = 4751_8$$
$$1111110000_2 = 001\,111\,110\,000_2 = 1760_8$$

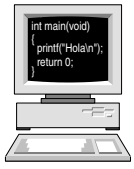
La conversión de octal a binario se realiza convirtiendo por separado cada dígito del número octal. Para ello puede ayudarse del cuadro 3.1. Por ejemplo:

$$4751_8 = 100\,111\,101\,001_2 = 100111101001_2$$

3.5.2. Conversión entre hexadecimal y binario

La conversión tanto de binario a hexadecimal como de hexadecimal a binario se realiza de la misma forma que entre binario y octal, salvo que se agrupan los bits de cuatro en cuatro. Así, para convertir de binario a hexadecimal se hace:

$$100111100111_2 = 1001\,1110\,0111_2 = 9E7_{16}$$



Realice el ejercicio 3

Y de hexadecimal a binario:

$$1CA1_{16} = 0001\,1100\,1010\,0001_2 = 0001110010100001_2 = 1110010100001_2$$

$a_i + b_i + c_i$			c_{i+1}	s_i
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

(a) Tabla de sumar

$a_i - (b_i + c_i)$			c_{i+1}	s_i
0	0	0	0	0
0	0	1	1	1
0	1	0	1	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0
1	1	0	0	0
1	1	1	1	1

(b) Tabla de restar

$a_i \times b_i$		s_i
0	0	0
0	1	0
1	0	0
1	1	1

(c) Tabla de multiplicar

Cuadro 3.2: Tablas de operaciones en binario

3.5.3. *Conversión de números con parte fraccionaria*

Si el número binario contiene una parte fraccionaria, para convertirlo a octal o a hexadecimal los grupos de bits se realizan a partir de la coma binaria. Para la parte fraccionaria puede ser necesario añadir ceros a la derecha para completar un grupo. Para ilustrar el método se muestran a continuación dos ejemplos, uno de conversión a octal y otro a hexadecimal.

$$10111011001,0111_2 = 010\,111\,011\,001\,,011\,100_2 = 2731,34_8$$
$$10111011001,0111_2 = 0101\,1101\,1001\,,0111_2 = 5D9,7_{16}$$

3.5.4. *Hexadecimal vs. octal*

En la práctica se usa principalmente el sistema hexadecimal frente al octal debido a que en los ordenadores los bits se agrupan en palabras con un número de bits múltiplo de 4.[6] Por ello, cualquiera de estos números se puede convertir a hexadecimal sin necesidad de añadir ceros por la izquierda.

3.6. Operaciones matemáticas con números binarios

Las operaciones en binario se realizan de la misma forma que en base 10. Es más, al disponer sólo de los símbolos 0 y 1, las tablas de sumar, restar y multiplicar son mucho más simples. El único problema es que al tener los números un mayor número de cifras las operaciones son más largas y tediosas.

3.6.1. *Suma de números binarios*

Para sumar dos números binarios el procedimiento es el mismo que para sumar números en base 10: se va sumando dígito a dígito, empezando por los dos dígitos menos significativos. Si en la suma se produce un acarreo, éste se suma a los

[6] Por ejemplo en un PC se pueden usar números de 8, 16, 32 y 64 bits.

dos siguientes bits. La tabla usada para realizar la suma es la mostrada en el cuadro 3.2(a), en donde a_i y b_i son los dígitos i de cada número, c_i es el acarreo que llega a esos dígitos, s_i es el resultado de la suma para el bit i y c_{i+1} es el acarreo que se envía a la cifra siguiente. Para ilustrar el proceso nada mejor que un ejemplo:

$$\begin{array}{rcccccccc} & {\scriptstyle 1} & & & {\scriptstyle 1} & {\scriptstyle 1} & & & \\ & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ + & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 \\ \hline & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Para empezar se suman los dos bits menos significativos: 0+0, y el resultado es 0 con acarreo 0. A continuación se suman los dos siguientes bits: $0 + 1$, obteniéndose como resultado un 1 con acarreo 0. Después se suma $1 + 1$ y se obtiene como resultado 0 con un acarreo de 1, A continuación se suman los dos siguientes bits más el acarreo anterior: $1 + 1 + 1$, obteniéndose un 1 como resultado y un 1 como acarreo. Como puede comprobar en el ejemplo el proceso continua así hasta el último dígito.

Desborde en la suma

En la sección 3.4 se ha comentado que en un sistema digital el número de bits para representar un número binario es fijo, por lo que sólo puede representarse un rango de números naturales. Cuando se realiza una suma de dos números el resultado suele representarse con el mismo número de bits que los operandos.[7] No obstante esto presenta un grave problema, ya que el resultado de una suma puede salirse del rango si ambos sumandos son suficientemente grandes. Por ejemplo el resultado de sumar $200 + 200$ con 8 bits en binario es:

$$\begin{array}{rcccccccc} & {\scriptstyle 1} & & & {\scriptstyle 1} & & & & \\ & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ + & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}$$

que como se puede apreciar necesita 9 bits para representarse. Si el resultado se trunca a 8 bits porque el sistema digital que se está usando es de 8 bits, el resultado obtenido será: $10010000_2 = 144$ que no se parece en nada al resultado real de 400. Por tanto en un sistema digital ha de detectarse cuándo se produce un desbordamiento para no tener en cuenta el resultado.[8]

[7] En la práctica no se realizan operaciones aisladas, sino una serie de cálculos en los cuales el resultado de una operación se utiliza como operando para la siguiente. Piense por ejemplo en un programa de ordenador. Si cada vez que se hace una suma hay que añadir un bit para garantizar que no se desborda el resultado, al final se acabaría con un elevado número de bits. Por tanto, lo más práctico es usar el mismo número de bits para todos los números dentro del sistema digital y hacer que dicho número de bits sea suficiente para representar todas las magnitudes con las que va a trabajar el sistema digital.

[8] Esto es lo que ocurre en una calculadora digital: si intenta sumar dos números muy grandes la calculadora muestra un error en lugar del resultado.

La forma de detectar el desbordamiento en la suma es comprobar si se produce un acarreo al sumar los dos bits más significativos. Si no se produce el resultado es correcto y si se produce es que ha ocurrido un desbordamiento.

3.6.2. Resta de números binarios

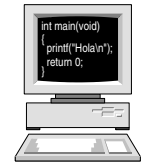
Al igual que la suma, el algoritmo de la resta es el mismo en binario que en decimal, sólo que ahora se usa la tabla 3.2(b). Al igual que con la suma el proceso es restar bit a bit empezando por los dos bits menos significativos. La principal diferencia con la suma es que el acarreo se le suma al sustraendo[9] y dicho resultado se resta del minuendo. El acarreo se produce tanto si al sumar el bit de acarreo al bit del sustraendo se produce un acarreo como si el bit del minuendo es un 0 y el resultado de $b_i + c_i$ es 1. Ilustremos el proceso con un ejemplo:

$$\begin{array}{r} 0\,1\,1\,0\,1\,1\,0\,0 \\ -\;\; 0\,1\,1\,0\,1\,0\,1\,1\,1\,0 \\ \hline 0\,0\,0\,1\,1\,1\,1\,0 \end{array}$$

En primer lugar se hace la resta 0 − 0, obteniéndose 0 como resultado y 0 como acarreo. Hasta aquí nada interesante. Para la siguiente cifra es necesario restar 0 − 1, con lo que se obtiene 1 como resultado y se produce un acarreo a la siguiente cifra. A continuación, en primer lugar se suma el acarreo a la cifra del minuendo, obteniéndose un 0 y produciéndose un acarreo a la siguiente cifra. En segundo lugar se resta 1 − 0 produciéndose 1 como resultado. Alternativamente a este proceso puede consultarse la tabla 3.2(b) para obtener el resultado de la resta de 1 − 1 con acarreo de entrada c_i igual a 1 (que es la última fila de dicha tabla). Repitiendo este proceso puede comprobar que se obtiene el resultado mostrado en el ejemplo.

Desborde en la resta

El algoritmo para restar mostrado sólo permite realizar la resta cuando el minuendo es mayor que el sustraendo. Por tanto en este caso nunca se producirá ningún error.



Realice el ejercicio 4

3.6.3. Multiplicación de números binarios

El algoritmo de la multiplicación no iba a ser menos: al igual que la suma y la resta, multiplicar números binarios se realiza de la misma manera que con números decimales. Además la tabla de multiplicar es mucho más sencilla en binario que en decimal, como se puede ver en 3.2(c). Por ejemplo, la multiplicación de 1010 por 1101 se realiza:

[9] Recuerde que si se hace la operación $a - b$, al número a se le denomina minuendo y al b sustraendo.

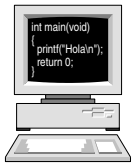
$$
\begin{array}{r}
\times\ 1\,0\,1\,0 \\
\hline
1\,1\,0\,1 \\
\hline
1\,0\,1\,0 \\
0\,0\,0\,0 \\
1\,0\,1\,0 \\
1\,0\,1\,0 \\
\hline
1\,0\,0\,0\,0\,0\,1\,0
\end{array}
$$

En primer lugar se multiplica el multiplicando por la primera cifra del multiplicador.[10] Como es un 1, el resultado es 1010. A continuación se multiplica el multiplicando por la segunda cifra del multiplicador, escribiendo el resultado desplazado una cifra a la izquierda. Como en el ejemplo la segunda cifra es un 0, la segunda fila es 0000. El algoritmo continua así hasta llegar a la última cifra del multiplicador. Para terminar se suman los cuatro productos parciales. Como puede observar la multiplicación en binario es mucho más fácil que en decimal, pues al ser las cifras del multiplicador 0 o 1 los productos parciales son o 0 o el multiplicando.

Desborde en la multiplicación

Como puede deducirse del algoritmo anterior, si se multiplican dos números de n bits, el resultado necesita $2 \times n$ bits para representarse. Si se trunca el resultado a n bits se producirá un desbordamiento si alguno de los n bits más significativos están a 1. En el ejemplo anterior se ha realizado la multiplicación 10×13, obteniéndose 130. Como puede observarse se han multiplicado dos números de 4 bits y el resultado ha necesitado 8 bits para representarse. Si sólo se dispusiera de 4 bits para el resultado, se habría obtenido un 2 como resultado, que obviamente es erróneo; lo cual se detecta viendo que hay bits a 1 en los 4 bits más significativos del resultado. Si en cambio se multiplica 7×2:

$$
\begin{array}{r}
\times\ 0\,1\,1\,1 \\
\hline
0\,0\,1\,0 \\
\hline
0\,0\,0\,0 \\
1\,1\,1 \\
0\,0\,0\,0 \\
0\,0\,0\,0 \\
\hline
0\,0\,0\,1\,1\,1\,0
\end{array}
$$



no se produce desbordamiento, cuando el resultado se representa con 4 bits.

Realice los ejercicios 5 y 6

3.7. Representación de números enteros

Para distinguir entre números positivos y negativos, nosotros usamos dos símbolos adicionales: el + para los positivos, omitido normalmente en la práctica, y el –

[10] En el ejemplo anterior el multiplicando es 1010 y el multiplicador 1101.

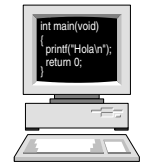
para los negativos. En un sistema digital, como se ha dicho anteriormente, sólo existen dos símbolos: el 0 y el 1; por lo que es necesario buscar formas de representar los números negativos sin usar los símbolos + y −.

3.7.1. Representación de números en signo-magnitud

La primera técnica que se nos viene a la cabeza para representar números con signo es usar un bit para representar el signo, asignando el 0 para el + y el 1 para el −. Este bit, al que se denomina **bit de signo**, se escribe al principio del número binario (bit más significativo). Así, si se dispone de 4 bits, se reserva el primero para el signo y los tres restantes para la magnitud. Por ejemplo, el +2 se representará en este caso como 0010 y el −2 como 1010.

A la vista del ejemplo anterior cabe preguntarse cómo sabe un sistema digital si la secuencia de bits 1010 es un −2 o un 10. La respuesta es que el sistema digital no lo sabrá nunca, pues no es un ser inteligente. Somos nosotros los encargados de interpretar cada secuencia de bits sabiendo de antemano cómo están codificados. En el ejemplo anterior, sabiendo que la secuencia de bits 1010 representa un número en signo magnitud podremos obtener su valor. Para ello, al ser el bit más significativo un 1 sabemos que es un número negativo. Para obtener la magnitud tomamos los bits restantes (010) y los convertimos a decimal, obteniendo un 2. Por tanto, si la secuencia de bits 1010 representa un número en signo-magnitud, dicho número es el -2.

La desventaja de este sistema de representación es que es complejo operar con él, tal como se discute en la sección 3.9.1. Por ello se han desarrollado otros métodos que aunque son menos intuitivos, facilitan mucho las operaciones.



Realice el ejercicio 7

3.7.2. Representación de números en complemento a dos

Esta técnica es la más usada en la práctica, pues como se verá más adelante simplifica las operaciones matemáticas entre números con signo. Esta representación se basa en obtener el complemento a dos de los números negativos, por lo que antes de empezar conviene mostrar cómo se obtiene el complemento a dos de un número.

Complemento a dos

Para obtener el complemento a dos de un número binario existen dos algoritmos:

- El primero consiste en invertir todos los bits del número binario y sumar uno al resultado. Por ejemplo, para obtener el complemento a dos de 0110, en primer lugar se invierten todos los bits:

  0 1 1 0
  1 0 0 1

  Al resultado obtenido se le suma 1:

$$\begin{array}{r} {\scriptstyle 1} \\ 1\,0\,0\,1 \\ +\;\; 0\,0\,0\,1 \\ \hline 1\,0\,1\,0 \end{array}$$

- El segundo método es mucho más fácil y consiste en copiar cada bit del número de derecha a izquierda hasta que se encuentre el primer 1. A partir de ese momento se copia el inverso de cada bit. En el ejemplo anterior, para obtener el complemento de 0110, se empieza copiando el número hasta encontrar el primer 1, con lo que en ese momento habremos escrito:

$$\begin{array}{r} 0\,1\,1\,0 \\ \hline \quad\;\; 1\,0 \end{array}$$

  El resto de bits se copian invertidos, obteniendo al final el complemento a dos del número:

$$\begin{array}{r} 0\,1\,1\,0 \\ \hline 1\,0\,1\,0 \end{array}$$

Ambos algoritmos son sencillos de realizar, pero quizás el lector no entienda en realidad qué es lo que se está haciendo. Pues bien, obtener el complemento a dos de un número x no es más que calcular $2^n - x$ en donde n es el número de bits del número x. Por ejemplo, para obtener el complemento a dos del número de 4 bits 0110 hay que hacer la siguiente operación:

$$\begin{array}{r} 1\,0\,0\,0\,0 \\ -\;\; 1\,0\,1\,1\,0 \\ \hline 0\,1\,0\,1\,0 \end{array}$$

Lo que ocurre es que la resta anterior es difícil de hacer, tanto para nosotros como para nuestros queridos circuitos digitales. Ahora bien, teniendo en cuenta que $10000 = 1111 + 1$, la anterior operación se puede descomponer en $1111 - 0110 + 1$. La ventaja de hacer esta operación así es que, tal como se aprecia en el ejemplo:

$$\begin{array}{r} 1\,1\,1\,1 \\ -\;\; 0\,1\,1\,0 \\ \hline 1\,0\,0\,1 \end{array}$$

restar $1111 - 0110$ es equivalente a invertir los bits del sustraendo. Esto es cierto siempre que el minuendo sea todo unos, pues no se producen acarreos entre cifras y $1 - 0 = 1$ y $1 - 1 = 0$.

Para terminar de calcular el complemento a dos se suma 1 al resultado anterior:

$$\begin{array}{r} {\scriptstyle 1} \\ 1\,0\,0\,1 \\ +\;\; 0\,0\,0\,1 \\ \hline 1\,0\,1\,0 \end{array}$$

En definitiva, con este truco hemos conseguido realizar la operación $2^n - x$ invirtiendo todos los bits y sumando 1 en lugar de hacer una resta.

Representación de números enteros en complemento a dos

Para representar un número entero en complemento a dos se sigue el siguiente algoritmo:

- Si el número es positivo, se representa en binario. En este caso el número de bits usados para representar el número ha de ser suficiente para que el bit más significativo sea un 0.
- Si el número es negativo se obtiene el complemento a dos del valor absoluto del número. En este caso el número de bits usados para representar el número ha de ser suficiente para que el bit más significativo sea un 1.

En ambos casos, al igual que en los números representados en signo-magnitud, al bit más significativo se le denomina **bit de signo** y es 0 cuando el número es positivo y 1 cuando es negativo.

Lo mejor como siempre es ver un ejemplo:

Representar los siguientes números en complemento a dos usando 4 bits:

- +3. Como es positivo se representa en binario sin más. Por tanto +3 se representa en complemento a 2 como 0011_{C2}.[11]
- -3. Como es negativo, ha de obtenerse el complemento a 2 de su módulo. Por tanto es necesario obtener el complemento a 2 de 0011_2, que es 1101_{C2}. Así pues -3 se representa en complemento a 2 como 1101_{C2}.

A veces es frecuente tener que realizar el proceso inverso, es decir, a partir de una representación en complemento a dos de un número, obtener su valor. Para ello basta con seguir el algoritmo presentado antes a la inversa. Para ilustrar el proceso nada mejor que realizar el siguiente ejercicio:

Ejercicio

Imagínese que está depurando un programa y en la memoria del ordenador lee dos posiciones de memoria en las que sabe que hay almacenados dos números en complemento a dos. Si los números en cuestión son:

- 00001011_{C2}
- 11111011_{C2}

¿De qué números se trata?

Para obtener el número entero representado por cada secuencia de bits, en primer lugar hay que averiguar si el número es positivo o negativo. Esto viene dado por el bit más significativo (bit de signo). Si dicho bit es un 0 el número es positivo, y si es un 1 es negativo. Si el número es positivo basta con convertir el número

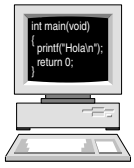
[11] En este libro, a los números representados en complemento a dos se les pondrá un C2 como subíndice.

binario a decimal. Así, como el primer número del ejemplo tiene el bit de signo a 0, éste es positivo y por tanto se trata del:

$$00001011_{C2} = 0\times2^7 + 0\times2^6 + 0\times2^5 + 0\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0 = 11$$

En cambio el segundo número tiene el bit de signo a 1, por lo que en primer lugar hay que obtener su complemento a dos:

$$\begin{array}{c} 1\ 1\ 1\ 1\ 1\ 0\ 1\ 1 \\ 0\ 0\ 0\ 0\ 0\ 1\ 0\ 1 \end{array}$$



El resultado anterior es el valor absoluto del número en binario, por lo que hay que convertirlo a decimal:

$$00000101_2 = 0\times2^7 + 0\times2^6 + 0\times2^5 + 0\times2^4 + 0\times2^3 + 1\times2^2 + 0\times2^1 + 1\times2^0 = 5$$

Por tanto el número 11111011_{C2} es el -5.

Realice los ejercicios 8 y 9

Todo esto está muy bien, pero... ¿Qué estamos haciendo?

La mecánica expuesta para representar números en complemento a dos puede parecer sin sentido, pero al contrario es una idea genial. Lo que se persigue con esta notación es facilitar la realización de sumas con números enteros. Como se ha dicho antes, en complemento a dos un número negativo $-x$ se representa como $2^n - x$. Para que la notación funcione, si se hace la operación $x + (-x)$ el resultado ha de ser cero. En la representación en complemento a dos esta operación se traduce en:

$$x + (2^n - x) = 2^n \tag{3.1}$$

Pero 2^n equivale a 0 si las operaciones se realizan con números de n bits y no se tiene en cuenta el acarreo. Por ejemplo, si $x = 0010_{C2}$, la operación anterior será, teniendo en cuenta que -2 en complemento a dos se representa como 1110_{C2}:

$$\begin{array}{rc} & {\scriptstyle 1\ 1\ \ \ \ } \\ & 0\ 0\ 1\ 0 \\ + & 1\ 1\ 1\ 0 \\ \hline & 1\ 0\ 0\ 0\ 0 \end{array}$$

Pero como tanto los operandos como el resultado son números son de 4 bits y no tenemos en cuenta el acarreo final, el resultado de la operación anterior es 0000_{C2}, como debe ser. Nótese que en esta operación el acarreo final no indica desborde, tal como ocurre en la suma sin signo mostrada en la sección 3.6.1. En la sección 3.9.2 se muestra cómo detectar el desbordamiento cuando se suman números con signo representados en complemento a dos.

3.7.3. *Representación de números en exceso*

Otra técnica usada en la práctica para representar números enteros consiste en sumar a todos los números una cantidad fija y representar en binario el resultado. La cantidad fija, denominada exceso o sesgo, puede ser o bien 2^{n-1} o bien $2^{n-1}-1$, en donde n es el número de bits usados para representar el número. En la práctica este tipo de representación se usa para los exponentes de los números en coma flotante, usándose como exceso el valor $2^{n-1}-1$.

Veamos algunos ejemplos con números de 8 bits tomando como exceso el valor: $2^{n-1}-1 = 2^7-1 = 127$:

- 27: 27 + 127 = 154. Convirtiendo 154 a binario se obtiene:

154	2							
0	77	2						
	1	38	2					
		0	19	2				
			1	9	2			
				1	4	2		
					0	2	2	
						0	1	2
							1	0

  y por tanto 27 en exceso 127 se representa como: $10011010_{\text{XS}-127}$.[12]
- -27: $-27 + 127 = 100$. Convirtiendo 100 a binario se obtiene 1100100_2. Por tanto -27 en exceso 127 se representa como: $01100100_{\text{XS}-127}$. Nótese que se ha añadido un 0 a la izquierda para obtener los 8 bits requeridos en la representación.
- -128: $-128 + 127 = -1$. Como -1 no puede representarse en binario, pues en binario puro sólo se pueden representar números naturales, -128 no puede representarse en exceso 127.

Realice el ejercicio 10

3.8. Rangos en los números con signo

El rango disponible para representar números con signo se reparte más o menos de igual forma para los números positivos y para los números negativos, aunque con ligeras diferencias en función del método usado para representarlos. Así, es fácil demostrar que el rango para n bits es igual a:

- $-2^{n-1}-1 \leftrightarrow 2^{n-1}-1$ para números codificados en signo-magnitud.
- $-2^{n-1} \leftrightarrow 2^{n-1}-1$ para números codificados en complemento a dos.
- $-2^{n-1}-1 \leftrightarrow 2^{n-1}$ para números codificados en exceso $2^{n-1}-1$.

[12] En este libro, a los números representados en exceso 127 se les pondrá XS – 127 como subíndice.

3.9. Operaciones matemáticas con números con signo

Las operaciones descritas en el apartado 3.6 sólo son válidas para números binarios naturales (sin signo). Para realizar operaciones de números enteros es necesario modificar los algoritmos según el tipo de codificación usada.

3.9.1. Operaciones con números en signo-magnitud

Para realizar una suma de dos números en signo-magnitud es necesario estudiar en primer lugar los signos de ambos números y a continuación hacer una suma o una resta en función de éstos. Por ejemplo si se desea hacer la siguiente operación entre dos números codificados en signo-magnitud con 4 bits:

$$0011 + 1010 \tag{3.2}$$

En primer lugar se observa que el segundo número es negativo (−2), por lo que la operación a realizar será:

$$0011 - 0010 = 0001 \tag{3.3}$$

Para realizar una resta el proceso es el mismo, por lo que no merece la pena emplear más tiempo en ello. Para multiplicar dos números el proceso es similar: se multiplican las magnitudes y el signo resultante se obtiene estudiando el signo de los operandos.

Aunque esta forma de operar no es complicada para nosotros, si lo es para un circuito. Por ello, tal como se mencionó en la sección 3.7.1, los números codificados en signo-magnitud prácticamente no se usan en la práctica.

3.9.2. Operaciones con números en complemento a dos

La notación en complemento a dos se desarrolló precisamente para simplificar las operaciones con números enteros. Como se ha mostrado en la sección 3.7.2, para sumar números codificados en complemento a dos se sigue el algoritmo de suma en binario y se desprecia el acarreo final cuando se produce. Por ejemplo, la suma 27 + (−36) con números de 8 bits es:

$$\begin{array}{rr} & {\scriptstyle 1\ \ 1} \\ & 0\,0\,0\,1\,1\,0\,1\,1 \\ + & 1\,1\,0\,1\,1\,1\,0\,0 \\ \hline & 1\,1\,1\,1\,0\,1\,1\,1 \end{array}$$

Y el resultado, teniendo en cuenta que es un número representado en complemento a dos, es −9, tal como era de esperar.

Otra de las ventajas de usar números en complemento a dos es que las restas se pueden hacer sumando el sustraendo cambiado de signo, no siendo necesario usar el algoritmo de resta que tan difícil suele resultar de aplicar. Es decir, suplantamos el algoritmo de resta por el de obtener el complemento a dos (muy fácil) y el de sumar (más fácil que el de restar). Esta simplificación no solo es buena para nosotros, sino

que también lo es para los circuitos digitales, tal como veremos más adelante. Por ejemplo, para restar 00001111 – 00000110, lo que en realidad hacemos es:

1. Obtenemos el complemento a dos del sustraendo: 11111010_{C2}
2. Sumamos el minuendo más el sustraendo:

$$\begin{array}{rr} & {\scriptstyle 1\ 1\ 1\ 1\ 1\ 1} \\ & 0\ 0\ 0\ 0\ 1\ 1\ 1\ 1 \\ + & 1\ 1\ 1\ 1\ 1\ 0\ 1\ 0 \\ \hline & 1\ 0\ 0\ 0\ 0\ 1\ 0\ 0\ 1 \end{array}$$

3. Despreciamos el acarreo final, con lo que el resultado es 00001001_{C2}, que es 9, como era de esperar después de restar 15 – 6.

Desborde en la suma en complemento a dos

Al igual que en la suma sin signo, si los números a sumar son demasiado grandes puede ocurrir que el resultado no pueda representarse con el número de bits finito usado para representarlo. Por ello después de realizar una suma es necesario verificar si se ha producido un desbordamiento para marcar el resultado como erróneo. Existen dos formas de evaluarlo, que como siempre es mejor discutir con un ejemplo: realicemos la suma de 103 + 103 en complemento a dos usando 8 bits:

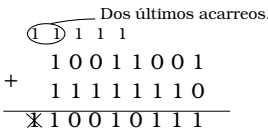
$$\begin{array}{rr} & {\scriptstyle 1\ 1\ \ \ \ 1\ 1\ 1} \\ & 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1 \\ + & 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1 \\ \hline & 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0 \end{array}$$

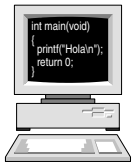
El resultado obtenido es 11001110_{C2}, que en decimal es −50 en lugar de 206, lo cual es una prueba de que algo no ha ido bien con la suma. El problema es que un sistema digital no es capaz de darse cuenta de este tipo de error si no buscamos una forma fácil de averiguarlo. Existen dos maneras:

- Si el signo de ambos sumandos es el mismo y el del resultado es distinto se ha producido un desbordamiento. Obviamente si después de sumar dos números positivos obtenemos uno negativo es señal de que algo no ha ido bien. De la misma forma si sumamos dos números negativos el resultado ha de ser otro número negativo y si nos sale positivo más vale que lo tiremos a la basura. Si el signo de ambos sumandos es distinto obviamente el resultado será menor que ambos sumandos, por lo que es imposible que se produzca un desbordamiento.
- Otra alternativa para detectar el desbordamiento es analizar los dos últimos acarreos. Si son iguales es señal de que ha ido todo bien, pero si son distintos es que ha ocurrido un desbordamiento. Si volvemos al ejemplo anterior:

$$\begin{array}{rr} & \overset{\text{Dos últimos acarreos.}}{\overbrace{}} \\ & {\scriptstyle 0\ 1\ 1\ \ \ \ 1\ 1\ 1} \\ & 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1 \\ + & 0\ 1\ 1\ 0\ 0\ 1\ 1\ 1 \\ \hline & 1\ 1\ 0\ 0\ 1\ 1\ 1\ 0 \end{array}$$

Vemos que los dos últimos acarreos son distintos, lo que indica que se ha producido un desbordamiento y por tanto el resultado no es correcto. Si en cambio se realiza la suma $-103 + (-2)$:





Realice el ejercicio 11

Los dos últimos acarreos son iguales y por tanto el resultado es correcto, lo cual puede comprobar usted mismo convirtiendo el resultado a decimal.

3.9.3. Extensión del signo

En ocasiones es necesario aumentar el número de bits con el que se representa un número para extender su rango. En este caso, si el número es sin signo basta con añadir ceros por la izquierda hasta completar el número de bits. Por ejemplo si tenemos el número 7 representado con 4 bits (0111) y queremos representarlo con 8 bits, basta con añadir cuatro ceros por la izquierda (0000 0111). Como puede comprobar fácilmente, el número resultante sigue siendo el 7.

Si queremos aumentar el número de bits de un número con signo representado en complemento a 2, el proceso es ligeramente distinto. Si el número es positivo, basta con añadir ceros por la izquierda, tal como acabamos de hacer con los números sin signo. Si el número es negativo la cosa cambia. Si partimos de un número cualquiera, como por ejemplo el -2 expresado con 4 bits (1110) y queremos expresarlo con 8 bits; vemos que si añadimos ceros por la izquierda obtenemos el número 0000 1110, que es el 14. Si por el contrario añadimos unos por la izquierda obtenemos el número 1111 1110, que como puede comprobar es el número -2.

Resumiendo, para aumentar el número de bits de un número en complemento a 2 basta con extender por la izquierda el bit de signo hasta completar el número de bits deseado. A este proceso se le conoce como **extensión de signo**.

Veamos algunos ejemplos:

- 0110 a 8 bits: 0000 0110
- 1101 a 16 bits: 1111 1111 1111 1101

3.9.4. Multiplicación de números en complemento a 2

Al igual que con la suma, el algoritmo de multiplicación también es válido con números en complemento a dos. No obstante antes de realizar la multiplicación es necesario extender el signo de ambos números para que las sumas intermedias se realicen siempre con el número de bits del resultado (que recuerde que es la suma de los números de bits de ambos operandos). Una vez realizada la multiplicación sólo se tienen en cuenta los bits menos significativos del resultado. Lo mejor es mostrar un ejemplo: si se desea multiplicar -2×5, estando ambos representados con 4

bits; el punto de partida es $1110_{C2} \times 0101_{C2}$. Como el resultado ha de expresarse con 8 bits, es necesario extender el signo de ambos operandos antes de realizar la multiplicación:

$$
\begin{array}{r}
 & 1\,1\,1\,1\,1\,1\,1\,0 \\
\times & 0\,0\,0\,0\,0\,1\,0\,1 \\
\hline
 & 1\,1\,1\,1\,1\,1\,1\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
 & 1\,1\,1\,1\,1\,1\,1\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
0\,0\,0\,0\,0\,0\,0\,0 & \\
\hline
0\,0\,0\,0\,1\,0\,0\,1\,1\,1\,1\,0\,1\,1\,0 &
\end{array}
$$

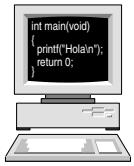
Del resultado de la multiplicación anterior sólo se toman los 8 bits menos significativos, obteniéndose entonces 11110110_{C2}, que como puede comprobar es -10.

Desborde en la multiplicación de números en complemento a 2

Al igual que con la multiplicación de números naturales, si el resultado se expresa con $n+m$ bits, siendo n y m el número de bits del multiplicando y el multiplicador; no se producirá ningún desbordamiento. Si se desea truncar el número de bits del resultado, habrá que verificar que el resultado está dentro del rango. Para ello basta con verificar que los bits que se eliminan del resultado son todos iguales al bit de signo del número resultante. En el ejemplo anterior, si queremos expresar el resultado original (11110110_{C2}) con 4 bits vemos que los 4 bits que se eliminan (1111) aunque iguales entre sí son distintos al bit de signo del número resultante (0110_{C2}). Por tanto se produce un desbordamiento, lo cual es fácil de comprobar, ya que 0110_{C2} es 6, en lugar de -10.

Si por ejemplo se realiza la multiplicación de -2×3 con 4 bits:

$$
\begin{array}{r}
 & 1\,1\,1\,1\,1\,1\,1\,0 \\
\times & 0\,0\,0\,0\,0\,0\,1\,1 \\
\hline
 & 1\,1\,1\,1\,1\,1\,1\,0 \\
 & 1\,1\,1\,1\,1\,1\,1\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
 & 0\,0\,0\,0\,0\,0\,0\,0 \\
0\,0\,0\,0\,0\,0\,0\,0 & \\
\hline
0\,0\,0\,0\,0\,1\,0\,1\,1\,1\,1\,1\,0\,1\,0 &
\end{array}
$$



Realice el ejercicio 12

El resultado en 8 bits es 11111010_{C2}, que en este caso se puede truncar a 4 bits, ya que los cuatro bits más significativos coinciden con el bit de signo del número truncado. El resultado es 1010_{C2} que como puede comprobar es -6.

3.10. Otros códigos binarios

A la hora de representar un número en binario lo primero que se nos viene a la cabeza es representar dicho número en base 2, tal como hemos hecho en las secciones anteriores. No obstante existen otros métodos para codificar números en binario que presentan ciertas ventajas.

3.10.1. Códigos BCD

La dificultad de convertir un número decimal a uno binario viene porque es necesario convertir el número como un todo, tal como hemos visto en la sección 3.3. Para facilitar la conversión, en los números en BCD se codifica cada uno de sus dígitos por separado. De ahí viene precisamente el nombre de BCD, que son las siglas del inglés *Binary Coded Decimal* o Decimal Codificado en Binario.

Así pues, para codificar un número en BCD se codifica en binario cada uno de los dígitos decimales que forman el número. Así, para codificar el número 127, lo que hacemos es codificar por un lado el 1, luego el 2 y luego el 7. Como en el sistema decimal tenemos 10 símbolos (del 0 al 9), son necesarios 4 bits para codificarlos. El problema es que con 4 bits tenemos 16 posibles combinaciones, de las cuales sólo vamos a usar 10. Esto hace que en total se puedan hacer $2,9 \cdot 10^{10}$ posibles asignaciones entre el código binario y el dígito decimal.[13] Afortunadamente en la práctica sólo se usan tres asignaciones, las cuales se muestran en el cuadro 3.3

En BCD natural se codifica cada dígito según su valor binario. El resto de códigos binarios (del 1010 al 1111) no se usan y por eso se han marcado con un guión en el cuadro. En el código Aiken los cinco primeros dígitos decimales se codifican según su valor, pero los otros 5 se codifican sumándole 6. Esto permite que la asignación de códigos binarios sea simétrica y facilita la operación matemática del complemento a 10, usada para representar números negativos al igual que el complemento a 2 de los números binarios. El código XS-3 consigue la misma simetría que el Aiken pero sumando 3 a todos los valores.

Para ilustrar el proceso de codificación, nada mejor que ver algunos ejemplos:

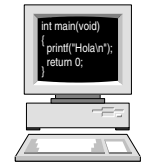
- Codificar 127 en BCD Natural, BCD Aiken y BCD XS-3. Usando el cuadro 3.3 codificamos cada dígito por separado:
  - BCD Natural: 0001 0010 0111.
  - BCD Aiken: 0001 0010 1101.
  - BCD XS-3: 0100 0101 1010.

[13] Las posibles asignaciones se calculan como las variaciones de 16 elementos tomados de 10 en 10, que vienen dadas por la fórmula $16!/(16-10)!$

$B_3B_2B_1B_0$	BCD Natural	BCD Aiken	BCD XS-3
0 0 0 0	0	0	–
0 0 0 1	1	1	–
0 0 1 0	2	2	–
0 0 1 1	3	3	0
0 1 0 0	4	4	1
0 1 0 1	5	–	2
0 1 1 0	6	–	3
0 1 1 1	7	–	4
1 0 0 0	8	–	5
1 0 0 1	9	–	6
1 0 1 0	–	–	7
1 0 1 1	–	5	8
1 1 0 0	–	6	9
1 1 0 1	–	7	–
1 1 1 0	–	8	–
1 1 1 1	–	9	–

Cuadro 3.3: Códigos BCD

- Convertir los siguientes números BCD a decimal. El proceso a seguir es el inverso del ejemplo anterior, teniendo en cuenta qué código BCD usa el número:
  - 1001 0110 0011, teniendo en cuenta que está codificado en BCD Natural: 963.
  - 0100 1011, teniendo en cuenta que está codificado en BCD Aiken: 45
  - 0100 1000 1100 1011, teniendo en cuenta que está codificado en BCD XS-3: 1598



Realice los ejercicios 13 y 14

Números BCD con signo

En BCD los números con signo se representan en la práctica o en signo magnitud o en complemento a 10. Ambos métodos son similares a los expuestos para números binarios.

Signo magnitud

Al igual que en binario, se reserva el dígito más significativo para el signo (0001 para – y 0000 para +). Así, el número +27 se representa en BCD natural como 0000 0010 0111 y el -27 como 0001 0010 0111.

Complemento a 10

El complemento a 10 de un número x se calcula del mismo modo que el complemento a 2 de los números binarios: $10^n - x$, siendo n el número de dígitos del número x. Por ejemplo el complemento a 10 del número 127 se calcula como:

$$\begin{array}{rr} & 1000 \\ - & 127 \\ \hline & 873 \end{array}$$

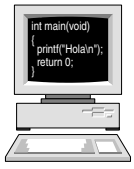
Por tanto, el complemento a 10 del número BCD natural 0001 0010 0111 es el 1000 0111 0011.

Para codificar números enteros en BCD usando complemento a 10 el proceso es el mismo al seguido con los números binarios:

- Si el número es positivo, se codifica dicho número en BCD. En este caso hay que asegurarse que el dígito mas significativo sea menor que 5.
- Si el número es negativo, se codifica en BCD el complemento a 10 del número. En este caso el dígito más significativo ha de ser mayor o igual que 5.

Por ejemplo, si queremos codificar el número +27 en BCD natural usando dos dígitos, como es positivo, se codifica tal cual: 0010 0111. Al ser el dígito más significativo menor que 5 es señal de que el resultado es correcto.

Para codificar el número -27 en BCD natural usando dos dígitos, como es negativo, es necesario codificar su complemento a 10, que es $100 - 27 = 73$. Por tanto -27 se codifica como 0111 0011. Nuevamente, como el dígito más significativo es mayor que cinco, el resultado es correcto.



Si nos dan un número entero en BCD codificado en complemento a 10 y queremos saber de qué número se trata, basta con realizar el algoritmo anterior a la inversa, tal como hicimos con los números binarios en complemento a 2.

En conclusión, como ha podido observar, el complemento a 10 en los números BCD es igual que el complemento a 2 en los números binarios.

Realice los ejercicios 15 y 16

Suma de números en BCD

El algoritmo para sumar números en BCD es ligeramente distinto al usado para sumar números en binario. Además dicho algoritmo varía en función del código BCD usado. Para sumar números codificados en BCD natural o BCD Aiken, el algoritmo es el siguiente:

- Se suma el número dígito a dígito.
- Para cada dígito, si el resultado es un código inválido (los marcados con un guión en la tabla 3.3) o si se produce un acarreo a la siguiente cifra, hay que sumar 6 al resultado anterior.

Para ilustrar el proceso nada mejor que mostrar un ejemplo. A continuación se realiza la suma 197 + 385 = 582 en BCD natural:

0011	0011	1110
0001	1001	0111
0011	1000	0101
0101	0010	1100
	0110	0110
	1000	0010

La suma del primer dígito da como resultado 1100, que como puede apreciar en la tabla 3.3 es un código inválido. Por tanto se ha sumado 6 al resultado (se han omitido los acarreos intermedios para no complicar la figura) para obtener un 2, que es el resultado esperado. Nótese que como resultado de esta última suma se produce un acarreo al siguiente dígito, el cual ha de tenerse en cuenta. La suma del segundo dígito da como resultado 0010, que es un código válido, pero como se ha producido un acarreo al siguiente dígito, es necesario sumar 6 al resultado anterior; obteniendo un 8 que de nuevo es el resultado esperado. Por último, en la tercera cifra el resultado es 0101 que al ser un código válido y al no haberse producido acarreo a la siguiente cifra no hace falta realizarle ningún ajuste.

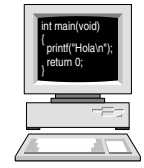
El algoritmo para números BCD XS-3 es ligeramente distinto y un poco más complejo al tener que realizar restas:

- Se suma el número dígito a dígito.
- Para cada dígito, si hay acarreo a la siguiente cifra se suma 3. Si no hay acarreo a la siguiente cifra se resta 3.

Por ejemplo, la suma 197 + 485 = 682 en BCD XS-3 se realiza:

	1111		1111		0000
	0100		1100		1010
	0111		1011		1000
	1100		1000		0010
-	0011	+	0011	+	0011
	1001		1011		0101

Como puede observar, en los dos primeros dígitos la suma ha producido un acarreo a la siguiente cifra, por lo que se ha sumado 3 al resultado. Por el contrario, en el último dígito no se ha producido ningún acarreo, por lo que se ha restado 3 al resultado. El número final ha sido 1001 1011 0101, que si lo convierte a decimal obtendrá 682, como era de esperar.



Realice el ejercicio 17

3.10.2. Código Gray

La característica del código Gray es que entre dos números consecutivos sólo cambia un bit. Si se fija en los diagramas de Karnaugh, las filas y columnas están numeradas según este código (00, 01, 11 y 10) para conseguir que entre casillas adyacentes sólo difiera un bit. La otra aplicación práctica de los números en código Gray son los codificadores angulares. Estos dispositivos permiten medir ángulos mediante unos sensores luminosos y una rueda codificada según el código Gray,

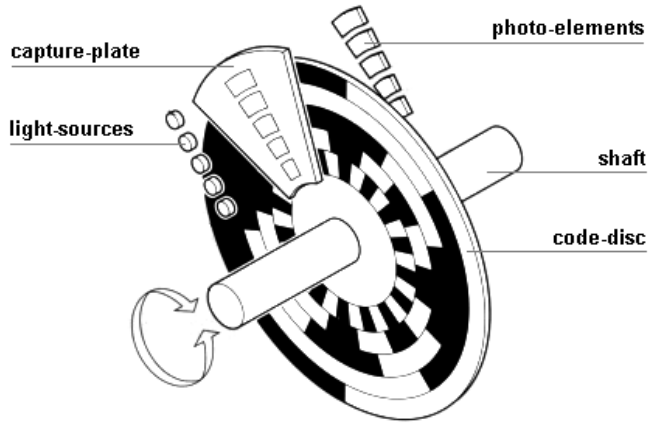


Figura 3.1: Codificador angular.

tal como se muestra en la figura 3.1. El funcionamiento de estos dispositivos es el siguiente: unos diodos led (*light-sources* en la figura) iluminan un disco dividido en sectores, cada uno de los cuales tiene impreso un número Gray (0 blanco, 1 negro). El disco por tanto dejará pasar la luz sólo en los puntos que estén en blanco, los cuales serán detectados por unos fototransistores (*photo-elements* en la figura). La salida de estos fototransistores será por tanto el número impreso en el disco en el sector que está frente a los detectores. El codificar los sectores en código Gray permite que cuando el disco está entre un sector y el siguiente no se produzcan errores en la lectura, ya que sólo cambiará un bit. Si por el contrario el disco se codificase en binario, existirían sectores consecutivos entre los que cambiaran varios bits. En ese caso, al pasar se un sector a otro, como es imposible que cambien varios bits a la vez, pasaremos por valores erróneos. Por ejemplo, si el disco estuviese codificado con tres bits, al pasar por ejemplo del sector 011 al 100 puede ocurrir que primero cambie el bit 2 y leamos el código 111, luego el bit 0 y leamos el código 110 y por último cambie el bit 1 para obtener el ángulo correcto 100. Si en cambio el disco está codificado en Gray, del 011 pasaremos al número 010 y como sólo cambia el bit 0 no podemos pasar por valores intermedios erróneos.

Conversión entre binario y código Gray

El algoritmo para convertir un número binario a Gray es bien sencillo:

- El bit más significativo del número en código Gray coincide con el del número binario.
- Para el resto de bits, el bit i del número en código Gray es igual a la XOR entre los bits i e $i+1$ del número binario.

Por ejemplo, para el número binario $b_3b_2b_1b_0$ el número en código Gray $g_3g_2g_1g_0$ se calcula como:

$$\begin{aligned} g_3 &= b_3 \\ g_2 &= b_3 \oplus b_2 \\ g_1 &= b_2 \oplus b_1 \\ g_0 &= b_1 \oplus b_0 \end{aligned}$$

Por ejemplo, el número binario 0100 se codifica en código Gray como 0110.

El algoritmo para convertir de Gray a binario es también muy simple:

- El bit más significativo del número binario coincide con el del número en código Gray.
- Para el resto de bits, si el bit i del número en código Gray es 1, el bit i del número binario es el bit $i+1$ del número binario negado. Si por el contrario el bit i del número en código Gray es 0, el bit i del número binario es el bit $i+1$ del número binario sin negar.

El algoritmo anterior es en realidad una forma práctica de calcular una serie de operaciones XOR encadenadas. Así, la expresión matemática para convertir el número en código Gray $g_3g_2g_1g_0$ a el número binario $b_3b_2b_1b_0$ es:

$$\begin{aligned} b_3 &= g_3 \\ b_2 &= g_3 \oplus g_2 \\ b_1 &= g_3 \oplus g_2 \oplus g_1 \\ b_0 &= g_3 \oplus g_2 \oplus g_1 \oplus g_0 \end{aligned}$$

Así, el número en código Gray 0110 se codifica en binario como 0100.

Realice el ejercicio 18

3.11. Ejercicios

1. Convierta los siguientes números de binario a decimal:
   - 10011101_2
   - 11101000_2
   - $101,1011_2$
   - $1101,001_2$
2. Convierta el número 23,57 a binario usando 6 bits para la parte decimal. Calcule también el error de cuantización cometido.
3. Convierta el número $1001010111101,01_2$ a hexadecimal y a octal.
4. Realice las siguientes operaciones en binario usando 8 bits: $78+76$, $78+200$, $78-60$.
5. Justifique el algoritmo de multiplicación expuesto en la sección 3.6.3. Para ello tenga en cuenta la definición de un sistema de numeración posicional expuesto en la sección 3.2.

6. Multiplique 14×11 en binario usando números de 4 bits. ¿Es posible representar el resultado también con 4 bits?
7. Represente +77 y -77 en binario signo-magnitud usando 8 bits.
8. Represente los números +77, -77, +128 y -128 en complemento a 2 usando 8 bits.
9. Si las siguientes secuencias de bits representan números en complemento a 2, ¿qué números representan?: 00100110_{C2}, 10010111_{C2}, 11111111_{C2}, 01010111_{C2}.
10. Codifique +28 y -28 con 8 bits usando signo-magnitud, complemento a 2 y XS-127.
11. Realice las siguientes operaciones, indicando cuándo el resultado es correcto y cuándo se produce un desbordamiento. Todas las operaciones son con números codificados en complemento a dos de 4 bits:
    a) $1001_{C2} + 0111_{C2}$
    b) $0101_{C2} + 1111_{C2}$
    c) $0111_{C2} + 0001_{C2}$
    d) $1100_{C2} + 1011_{C2}$
12. Realice las siguientes multiplicaciones usando números en complemento a 2 de 4 bits. Indique cuando el resultado puede expresarse sólo con 4 bits.
    a) $1001_{C2} \times 0111_{C2}$
    b) $0101_{C2} \times 1111_{C2}$
    c) $0111_{C2} \times 0010_{C2}$
    d) $1110_{C2} \times 0011_{C2}$
13. Convierta los siguientes números a BCD Natural, BCD Aiken y BCD XS-3:
    a) 1267
    b) 45
    c) 3987
14. Convierta los siguientes números de BCD a decimal.
    a) 0001 1000 1001 0110, sabiendo que está codificado en BCD Natural.
    b) 0011 1100 1110 0000, sabiendo que está codificado en BCD Aiken.
    c) 0100 0111 1100 0011, sabiendo que está codificado en BCD XS-3.
15. Codifique los siguientes números enteros en BCD natural en complemento a 10 usando números de 4 dígitos:
    a) 1267
    b) 45
    c) -3987
    d) -5678

16. Las siguientes secuencias de bits son números enteros codificados en BCD natural en complemento a 10. Indique qué numero decimal representan.

    a) 0001 1000 1001 0110.

    b) 0111 0000 0110 0010.

17. Realice la suma 398 + 582 en BCD natural. Para ello convierta ambos sumandos a BCD natural y realice la suma aplicando el algoritmo mostrado en la sección 3.10.1, indicando los acarreos intermedios. Repita el ejercicio codificando ambos números en BCD XS-3.

18. Convierta el número 1001 de código Gray a binario y el número 0111 de binario a código Gray.

CAPÍTULO 4

Introducción al lenguaje VHDL

En este capítulo se introduce el lenguaje de descripción de *hardware* VHDL. Estudiaremos cómo especificar circuitos con el lenguaje y cómo escribir bancos de prueba para verificar el funcionamiento mediante simulación.

4.1. Introducción

El lenguaje VHDL se desarrolló a mediados de los años 80 en un proyecto financiado por el departamento de defensa de los Estados Unidos de América y el IEEE.[1] El objetivo del lenguaje era permitir la especificación de circuitos integrados de alta velocidad y de ahí precisamente es de donde viene el nombre VHDL, que es un acrónimo de *VHSIC Hardware Description Language*, en donde VHSIC es otro acrónimo de *Very High Speed Integrated Circuit*.[2]

La utilidad de un lenguaje de especificación de *hardware* es poder construir descripciones de circuitos que sean inequívocas. Si se usa un lenguaje natural es fácil que se produzcan imprecisiones en la descripción, que al ir pasando de ingeniero en ingeniero generen errores al interpretar cada uno la especificación de una manera. Sin embargo, al usar un lenguaje como VHDL, la descripción se hace tan inambigua que hasta un programa de ordenador es capaz de simularla y verificar el funcionamiento del circuito. Así pues, el objetivo inicial del lenguaje VHDL fue precisamente este: poder especificar un circuito sin ambigüedades para que se pudiera simular.

Aunque poder documentar y simular un circuito era ya un avance, lo que ya fue un avance increíble fue el desarrollo de **sintetizadores** que a partir de una especificación en VHDL eran capaces de generar un circuito automáticamente. Esto ha permitido un aumento en la productividad de los diseñadores, ya que como veremos más adelante, mediante unas cuantas líneas de código VHDL seremos capaces de generar circuitos con miles de puertas lógicas.

Existen otros lenguajes de descripción de *hardware*, siendo Verilog el más popular. Curiosamente, en U.S.A se usa más Verilog y en Europa se usa más VHDL, a

[1] Institute of Electrical and Electronic Engineers.

[2] Por si no se ha dado cuenta aún, a los ingenieros nos encantan los acrónimos, y eso de que un acrónimo contenga otro es ya el no va más.

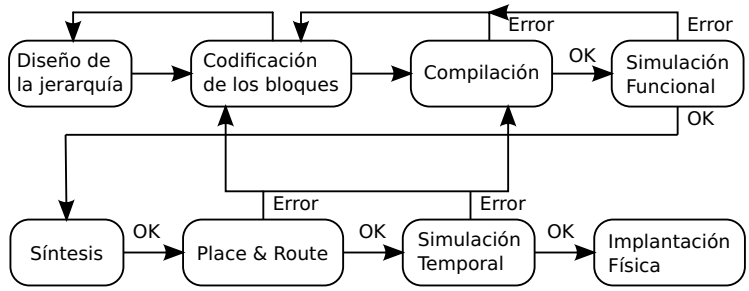


Figura 4.1: Flujo de diseño usando HDL.

pesar de ser un lenguaje desarrollado por el departamento de defensa americano. No obstante, aunque en este curso vamos a estudiar VHDL, no tiene por qué preocuparse si algún día tiene que usar Verilog. Lo difícil es aprender a describir *hardware* con un HDL.[3] Una vez aprendido, hacerlo con un lenguaje nuevo es sólo cuestión de familiarizarse con una nueva sintaxis.

4.2. Flujo de diseño

El diseño de un circuito incluye una serie de pasos, tal como se muestra en la figura 4.1. En esta sección se muestran las distintas etapas que componen el proceso de diseño y cómo se relacionan entre sí.

4.2.1. Diseño de la jerarquía del circuito

El proceso de diseño de un circuito complejo comienza dividiéndolo en bloques jerárquicos, de forma que conforme vamos bajando en la jerarquía tenemos bloques más simples. La jerarquía termina cuando los bloques son lo suficientemente simples como para ser descritos con unas pocas líneas de código. En cierto modo el proceso es el mismo que el usado en el desarrollo de un programa de ordenador.

Las ventajas del diseño jerárquico son numerosas. Las más importantes son:

- Cada bloque tiene una complejidad razonable.
- Cada bloque puede simularse por separado. Esto permite un modo de trabajo consistente en diseñar un bloque, simularlo y no pasar al siguiente hasta que estemos seguros de que funciona. Esto nos permite ir diseñando el circuito poco a poco, lo cual nos facilitará mucho la vida.
- Si se divide el circuito adecuadamente, especificando claramente qué hace cada uno de los bloques y su interfaz con el exterior (entradas y salidas) se puede repartir el diseño entre varios ingenieros. O acaso piensa que circuitos tan complejos como un microprocesador son el trabajo de una sola persona.

[3] *Hardware Description Language*.

- Si se diseñan bloques suficientemente genéricos se pueden reutilizar en otras partes del proyecto o incluso en otros proyectos. Es más, incluso podemos adquirir bloques ya diseñados, denominados *cores*, para incluirlos en nuestros circuitos o usar *cores* de código abierto.[4]

4.2.2. Codificación de los bloques en HDL

Una vez diseñada la jerarquía, se pasa a codificar cada uno de los bloques. Esta codificación la puede realizar con su editor de texto favorito, o bien puede usar el editor que traen los sistemas de desarrollo como Quartus II. La ventaja de estos últimos es que incluyen plantillas con las estructuras de código más usuales, generan bancos de prueba automáticamente, permiten llamar al compilador con una pulsación de ratón etc.

Tal como se muestra en la figura 4.1, a veces es necesario volver atrás en el flujo de diseño, por ejemplo cuando nos damos cuenta que el bloque que queremos diseñar es demasiado complejo o necesita alguna entrada o salida que no hemos tenido en cuenta al diseñar la jerarquía. No obstante, en general, estas iteraciones en el proceso de diseño no suelen consumir demasiado tiempo.

4.2.3. Compilación

Una vez codificado un bloque, es conveniente compilarlo para verificar que no hay errores sintácticos. El compilador además genera una serie de ficheros con la información del circuito especificado que son el punto de partida del simulador y del resto de herramientas del flujo de diseño.

Los errores en esta etapa normalmente se solucionan fácilmente volviendo a editar el código para corregirlos.

4.2.4. Simulación funcional

Cuando el código en VHDL compila sin errores podemos pasar a una primera simulación, denominada simulación funcional. En este primer paso sólo se tiene información del funcionamiento lógico del circuito, pues al no haber realizado la síntesis lógica no sabemos con qué puertas se va a implantar el circuito y por tanto no tenemos información acerca de los retardos de las señales dentro del circuito. La ventaja de esta simulación es que es muy rápida, por lo que el ciclo de simulación, detección de errores, edición del código VHDL para corregirlos, compilación y vuelta a simular es muy rápido.

4.2.5. Síntesis y Place & Route

Cuando la simulación funcional es correcta se puede pasar ya a sintetizar el circuito. En este paso el sintetizador genera el circuito especificado mediante VHDL,

[4] En este sentido la idea es la misma que la del *software* libre. Un repositorio muy conocido de *hardware* libre es *opencores*: www.opencores.org

teniendo en cuenta la tecnología en la que se va a implantar. Por ejemplo, si se va a implantar con una CPLD, el sintetizador genera funciones lógicas del tipo AND–OR. Si en cambio se va a implantar en un ASIC, el sintetizador tiene una mayor libertad a la hora de generar el circuito lógico.

Una vez generado el esquema lógico, el siguiente paso es decidir cómo se van a colocar las puertas en el circuito (*Place*) y cómo se van a cablear (*Route*).

Ambos procesos son computacionalmente intensivos, por lo que cuando el circuito es complejo tardan un buen rato. Por ello sólo pasamos a esta etapa cuando la simulación funcional es satisfactoria.

En algunos casos puede ocurrir que esta etapa falle, por ejemplo si queremos implantar el circuito en una FPGA que no tiene el número suficiente de puertas lógicas. Este error se corregirá o bien cambiando la FPGA por una más grande o bien jugando con las opciones del compilador para que genere un circuito más pequeño (aunque sea más lento) o incluso cambiando el código VHDL para especificar un circuito más simple.

4.2.6. Simulación temporal

Al finalizar la etapa anterior ya conocemos los retardos de todas las señales del circuito, por lo que podemos realizar una simulación muy exacta del circuito final y verificar que el dicho circuito sigue funcionando a pesar de los retardos de las señales. Si no es así, al igual que antes, las alternativas son o cambiar de tecnología de implantación (por ejemplo una FPGA más rápida), guiar al sintetizador para que intente generar circuitos más rápidos o cambiar el código VHDL para especificar otro circuito que cumpla con las especificaciones temporales. En cualquiera de los casos, los errores detectados en esta etapa son bastante más costosos de arreglar; aparte de que cada iteración lleva mucho tiempo al tener que volver a repetir el proceso de compilación, síntesis y *place & route*.

4.2.7. Implantación física

Una vez que el circuito se ha simulado extensivamente ya se puede dar el paso definitivo y fabricarlo. Tenga en cuenta que los costes no retornables de un ASIC son enormes, por lo que hemos de estar seguros de que va a funcionar a la primera.

Si el circuito se va a implantar en una FPGA, un fallo en la implantación no es tan crítico. No obstante, en general es mejor realizar la depuración del circuito mediante simulación, pues el proceso de descarga lleva su tiempo y sobre todo porque en simulación podemos ver todas las señales internas del circuito e interactuar fácilmente con él. En cambio en el circuito físico sólo tenemos acceso a las salidas y a través del osciloscopio, por lo que es muy difícil buscar los posibles fallos.

4.3. Estructura del archivo

Un archivo en VHDL contiene la descripción de un bloque jerárquico. Por ejemplo, el archivo siguiente contiene la descripción del multiplexor de 2 a 1 estudiado

en el capítulo 1.

```
1 -- Mutiplexor de 2 a 1 de 1 bit.
2
3 library IEEE;
4 use IEEE.std_logic_1164.all;
5
6 entity Mux2 is
7   port(a0, a1 : in  std_logic;      -- Entradas de datos
8        sel    : in  std_logic;      -- Entrada de selección
9        z      : out std_logic);     -- Salida seleccionada
10 end Mux2;
11
12 architecture RTL of Mux2 is
13 begin
14   z <= a0 when sel = '0' else a1;
15 end RTL;
```

La primera línea es un comentario que indica el contenido del archivo. En VHDL los comentarios comienzan con `--` y terminan en el final de la línea.

La línea 3 abre la librería IEEE y la 4 importa todas las definiciones del paquete `std_logic_1164`. Este paquete define el tipo `std_logic` que usaremos en todas las señales, por lo que habrá que incluirlo siempre. De momento piense en estas dos líneas como el equivalente al `#include <stdio.h>` de los programas en C.

Las líneas 6 a 10 definen el interfaz del bloque (**`entity`** en VHDL) con el exterior. El bloque ha de tener un nombre que ha de coincidir con el nombre del archivo. En este caso el nombre del bloque es `Mux2` y el archivo ha de llamarse por tanto `Mux2.vhd`. Las líneas 7 a 9 definen los puertos de entrada/salida que conectan el bloque con el exterior. En este caso el bloque dispone de tres entradas (`a0, a1` y `sel`) y una salida (`z`). Nótese que se ha indicado con un comentario un comentario la funcionalidad de las entradas y salidas.

Por último las líneas 12 a 15 definen el "interior" del bloque, lo que en VHDL se denomina la arquitectura. Para darle mayor flexibilidad al lenguaje se permite que un mismo bloque tenga varias arquitecturas y que en tiempo de compilación se elija la que se quiere usar. La utilidad de esto es poder tener un modelo simple para simular más rápido y un modelo complejo para sintetizar. Por ello toda arquitectura ha de tener un nombre (`RTL` en este caso). Nosotros sólo vamos a definir una arquitectura por cada bloque, pero aun así es obligatorio darle un nombre.

4.3.1. Sintaxis del interfaz del bloque (`entity`)

La sintaxis para definir la interfaz del bloque es la siguiente:

```
entity NombreDeEntidad is
  [ generic (declaraciones de parámetros); ]
```

```
  port (nombres de señales : modo tipo;
        nombres de señales : modo tipo;

        nombres de señales : modo tipo);
end [NombreDeEntidad];
```

La interfaz comienza con la palabra clave **entity** seguida del nombre que queramos darle a la entidad. Este nombre ha de ser representativo de la funcionalidad del bloque y ha de ser un identificador válido en VHDL. La línea se termina con la palabra clave **is**.

A continuación se pueden añadir de forma opcional[5] los denominados genéricos, que son constantes locales como tamaños de variables o temporizaciones que permiten definir componentes genéricos (de ahí el nombre). Como de momento no los vamos a usar, dejaremos su exposición para más adelante.

A los genéricos le sigue la declaración de los puertos del circuito. La declaración comienza con **port**(y termina con); y consiste en una serie de declaraciones de señales separadas por un punto y coma. Estas declaraciones constan del nombre de una o varias señales separadas por comas, seguidos del carácter : y un modo que indica la dirección de la señal seguido de un tipo de señal. El modo puede ser:

- **in** – Las señales son de entrada.
- **out** – Las señales son de salida.
- **inout** – Las señales son de entrada y salida.[6]

Por otro lado el tipo de la señal, aunque puede ser alguno de los tipos de datos válidos en VHDL, en la práctica sólo usaremos los tipos `std_logic` para señales de un bit y `std_logic_vector` para señales de varios bits.

Por ejemplo un bloque para comparar dos palabras de dos bits se declararía como:

```
entity Comparador2Bits is
  port( a, b : in std_logic_vector(1 downto 0);
        a_mayor_b, a_menor_b, a_igual_B : out std_logic);
end Comparador2Bits;
```

Identificadores en VHDL

Existen unas reglas básicas que han de cumplir los identificadores en VHDL:

- Ha de comenzar por una letra.
- Sólo debe de contener letras del alfabeto inglés, dígitos o el carácter de subrayado _. No está permitido que haya dos caracteres de subrayado seguidos ni que el último carácter sea el subrayado.

[5] Los elementos opcionales al lenguaje se mostrarán entre corchetes en este texto.

[6] Existe un cuarto modo denominado **buffer**, pero no es recomendable su uso para especificar circuitos, por lo que no se usa en el texto.

- No se distinguen mayúsculas de minúsculas.

Ejemplos de identificadores válidos en VHDL son: `Multiplexor`, `DetectorFlanco`, `entrada1`, `cuenta_arriba`. Ejemplos de identificadores erróneos son: `2entrada`, `_mi_variable_mala`, `otra__variable__mala_`, `variable_ñoña`. Por último, tenga en cuenta que a efectos del lenguaje, los identificadores `Hola`, `hola` y `HoLa` son el mismo.

Para terminar con los identificadores, es recomendable usar unas reglas de estilo que nos permitan al ver un identificador saber qué demonios está identificando. Así, los nombres de bloques comenzarán con una mayúscula. Si un bloque necesita varias palabras, se pondrán juntas empezando cada nueva palabra por mayúscula. Por ejemplo, `Mux2`, `MiBloque`, `MiOtroBloque` son identificadores recomendables para bloques. Por otro lado, los nombres de señales se escribirán en minúsculas y si el nombre es compuesto se separarán las palabras con el carácter de subrayado. Por ejemplo, `alarma` y `my_signal` son nombres de señal válidos.

En cualquier caso, el identificador ha de servir para hacernos una idea clara de lo que está identificando. Así no es recomendable llamar a los bloques de un circuito `Bloque1`, `Bloque2`, ... `Bloque5437`, pues a poco que tengamos unos cuantos bloques no sabremos para qué sirve cada uno. De la misma manera, las señales han de llamarse con nombres que nos recuerden su finalidad. Nombres de señal como `a` o `i` sólo son aceptables como contadores de bucles o alguna operación intermedia.

4.3.2. Sintaxis de la arquitectura

La sintaxis para definir la arquitectura de un bloque es la siguiente:

```
architecture NombreDeArquitectura of NombreDeEntidad is
  -- Declaraciones
begin

  -- Sentencias

end NombreDeArquitectura;
```

Como puede apreciar, la arquitectura comienza con la palabra clave **`architecture`** seguida del nombre elegido para la arquitectura. Aunque este nombre puede ser cualquiera,[7] se suelen usar los siguientes en función del tipo de descripción usada para especificar el circuito:

- `structural`. Se usa cuando la arquitectura se describe uniendo componentes entre sí. De este modo el lenguaje se usa de forma parecida a un editor gráfico.
- `behavioral`. Este nombre se usa cuando se describe la arquitectura en alto nivel.

A continuación viene la palabra clave **`of`** seguida del nombre de la entidad, que obviamente ha de coincidir con el que le hemos dado en la definición de la entidad y

[7] Siempre que sea un identificador válido en VHDL.

con el nombre del archivo VHDL. Después se escribe **is** y a continuación vienen las declaraciones de las señales internas que se necesiten en la arquitectura. Además de señales pueden declararse también constantes, funciones, tipos de datos nuevos, etc. No obstante todos ellos serán objetos locales a la arquitectura.

La arquitectura propiamente dicha se especifica mediante una serie de sentencias VHDL entre las palabras claves **begin** y **end**. Por último, después de **end** ha de escribirse obligatoriamente el nombre dado a la arquitectura.

4.4. Ejemplos

Veamos a continuación algunos ejemplos más de bloques especificados en VHDL. Empecemos por un viejo conocido: el circuito de alarma diseñado en el capítulo 2. Si recuerda el circuito disponía de una entrada de conexión para activar el sistema (*ON*) y de dos sensores, uno para detectar la apertura de la puerta (*P*) y otro la de la ventana (*V*). La ecuación del circuito expresada como suma de productos era:

$$Al = ON \cdot V + ON \cdot P \tag{4.1}$$

Si usamos nombres más descriptivos en el código, la representación del circuito en VHDL quedaría como:

```
-- Alarma casera. El circuito activa una señal de alarma
-- cuando el sistema esté conectado y se detecte la apertura
-- de una puerta o una ventana.

library IEEE;
use IEEE.std_logic_1164.all;

entity CircuitoAlarma is

  port (
    conexion : in  std_logic;   -- Conexión del sistema
    puerta   : in  std_logic;   -- Apertura de puerta
    ventana  : in  std_logic;   -- Apertura de ventana
    alarma   : out std_logic);  -- ¡Alarma. Llamen a la
                                -- Policía !
end CircuitoAlarma;

architecture behavioral of CircuitoAlarma is
  signal P1, P2 : std_logic;    -- Productos intermedios
begin  -- behavioral
  P1     <= conexion and ventana;
  P2     <= conexion and puerta;
  alarma <= P1 or P2;
end behavioral;
```

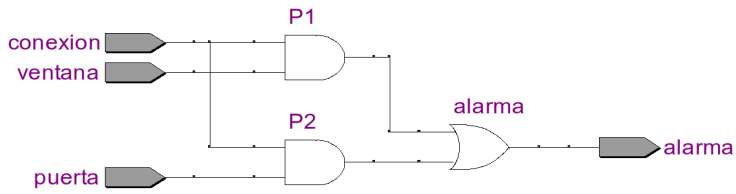


Figura 4.2: Resultado de la síntesis del circuito de alarma con Quartus II.

Nótese que hemos creado dos señales intermedias, P1 y P2 para contener los dos productos. El objetivo ha sido precisamente el de mostrar cómo se crean. Como puede observar, éstas se declaran entre el **is** y el **begin** y consisten en la palabra clave **signal** seguida del nombre de las señales, dos puntos, y el tipo de datos usado para la señal, std_logic en este caso.

Sentencias concurrentes

Aunque le parezca mentira, la arquitectura anterior se puede escribir también así:

```
1 architecture behavioral of CircuitoAlarma is
2   signal P1, P2 : std_logic;    -- Productos intermedios
3 begin  -- behavioral
4   alarma <= P1 or P2;
5   P1     <= conexion and ventana;
6   P2     <= conexion and puerta;
7 end behavioral;
```

Esto es debido a que el lenguaje está describiendo un circuito, que es precisamente el mostrado en la figura 4.2. La línea 4 está especificando que la salida alarma ha de conectarse a la salida de una puerta OR cuyas entradas son las señales P1 y P2. La línea 5 dice que la señal P1 es la salida de una AND cuyas entradas son conexion y ventana y la línea 6 dice que la señal P2 es la salida de otra AND cuyas entradas son conexion y puerta. Por tanto el orden en el que se escriban las sentencias es irrelevante.[8] Dicho de otro modo, podemos pensar que las sentencias en VHDL se "ejecutan" de forma concurrente, pues el *hardware* que están modelando es también concurrente. En un circuito todas las puertas funcionan a la vez y en cuanto cambia una de sus entradas actualizan su salida después de un pequeño retardo.

En cuanto al simulador, lo que hace es que cada vez que cambia una entrada busca por todo el código en qué sentencias interviene y calcula si se modifica alguna otra señal como consecuencia de este cambio. Si es así, repite el proceso para la

[8] No obstante veremos algunos casos en los que el orden de las sentencias sí que importa, pero esto será más adelante.

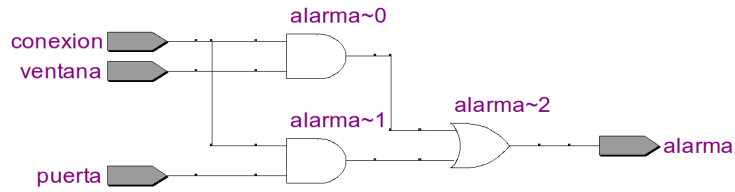


Figura 4.3: Resultado de la síntesis del circuito de alarma sin señales intermedias.

nueva señal que ha cambiado. Si no, se pone muy contento porque ya no tiene que hacer nada más hasta que cambie otra entrada.

Sin señales intermedias

La misma arquitectura anterior se puede especificar prescindiento de las señales intermedias, que recuerde que las pusimos en el ejemplo anterior más que por su necesidad, por mostrar cómo se declaraban y usaban.

```
1 architecture behavioral of CircuitoAlarma is
2 begin  -- behavioral
3   alarma <= (conexion and ventana) or (conexion and puerta);
4 end behavioral;
```

Como puede observar esta descripción se asemeja a la ecuación lógica de la ecuación 4.1, por lo que es más fácil de escribir y de interpretar. Lo que perdemos en este caso es la pista a las señales intermedias, tal como se observa en la figura4.3.

4.4.1. Descripción estructural

Una arquitectura también se puede definir como la interconexión de una serie de bloques. Así es precisamente como se definen los niveles superiores de la jerarquía. Para mostrar este tipo de descripciones vamos a repetir el ejemplo anterior usando un bloque para calcular la AND de dos señales y así generar los dos productos P1 y P2. El diseño está dividido en dos archivos VHDL. El primero es el bloque que contiene la puerta AND, denominado `PuertaAnd.vhd`:

```
-- Este bloque define una puerta AND de dos entradas.

library IEEE;
use IEEE.std_logic_1164.all;

entity PuertaAnd is
```

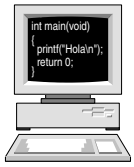
```
  port (
    a, b : in  std_logic;                -- Entradas
    s    : out std_logic);               -- Salida

end PuertaAnd;

architecture behavioural of PuertaAnd is

begin  -- behavioural

  s <= a and b;

end behavioural;
```

Como puede observar no hay nada nuevo en esta descripción. En cambio, la descripción del circuito de alarma es ahora totalmente distinta, al usarse una descripción estructural para instanciar las dos puertas AND descritas en el bloque anterior. El archivo `CircuitoAlarma2.vhd` queda:

```
1  -- Alarma casera. El circuito activa una señal de alarma
2  -- cuando el sistema está conectado y se detecte la apertura
3  -- de una puerta o una ventana.
4
5  library IEEE;
6  use IEEE.std_logic_1164.all;
7
8  entity CircuitoAlarma2 is
9
10   port (
11     conexion : in  std_logic;   -- Conexión del sistema
12     puerta   : in  std_logic;   -- Apertura de puerta
13     ventana  : in  std_logic;   -- Apertura de ventana
14     alarma   : out std_logic);  -- ¡Alarma. Llamen a la
15                                 -- Policía !
16 end CircuitoAlarma2;
17
18 architecture structural of CircuitoAlarma2 is
19   -- Declaraciones de señales
20   signal P1, P2 : std_logic;    -- Productos intermedios
21   -- Declaraciones de componentes
22   component PuertaAnd
23     port (
24       a, b : in  std_logic;           -- Entradas
25       s    : out std_logic);          -- Salida
26   end component;
```

```
27 begin  -- structural
28   i1: PuertaAnd
29     port map (
30       a => conexion,
31       b => ventana,
32       s => P1);
33
34   i2: PuertaAnd
35     port map (
36       a => conexion,
37       b => puerta,
38       s => P2);
39
40   alarma <= P1 or P2;
41 end structural;
```

Ahora en las líneas 22 a 26 se declara el componente `PuertaAnd`, pues es necesario declararlo antes de usarlo. Como puede observar la declaración del componente para ser usado en una arquitectura es muy similar a la declaración de la entidad. Tan solo se cambia **`entity`** por **`component`**, se elimina el **`is`** y en el **`end`** en lugar de poner el nombre de la entidad se pone **`component`**. Por tanto, para declarar un componente el *copy&paste* será su aliado.



Realice el ejercicio 1

Por otro lado, para instanciar un componente en la arquitectura es necesario darle un nombre de instancia seguido de dos puntos (`i1:` e `i2:` en el ejemplo) seguido del nombre del componente. A continuación mediante la sentencia **`port map`** se conectan las señales de la arquitectura a los puertos del componente. Por ejemplo, en la instancia `i1` de la puerta AND se ha conectado la señal `conexion` a la entrada `a` de la puerta, la señal `ventana` a la entrada `b` y la señal `P1` a la salida de la puerta.

4.5. Tipos de datos, constantes y operadores

Las señales y constantes en VHDL han de tener un tipo de datos asociado. Este tipo de datos define el rango de valores válidos, el conjunto de operadores que se pueden aplicar a los datos, etc.

Por otro lado el VHDL es un lenguaje de “tipado” fuerte, lo cual es una manera educada de decir que si intentamos asignar un tipo de dato a otro distinto el compilador nos mandará a paseo. No obstante no se preocupe, que existen funciones de conversión entre tipos.

En VHDL existen tipos de datos predefinidos por el lenguaje y tipos definidos por el usuario. De éstos últimos, algunos como el `std_logic` han sido definidos por organizaciones de estandarización como el IEEE y son ampliamente usados en la industria.

Tipos de datos predefinidos

A continuación se muestran los tipos de datos predefinidos por el lenguaje. Los que usaremos con mayor frecuencia son los tres primeros. Los cuatro siguientes se usan con menor frecuencia y los dos últimos no los usaremos nunca.

- `character`. Almacena un carácter codificado en ASCII.
- `integer`. Almacena un entero con signo de 32 bits.
- `time`. Almacena, como su propio nombre indica, un tiempo. Nosotros lo usaremos para generar retardos en los *testbench*. También es usado por los fabricantes para especificar los retardos de los circuitos para que se puedan realizar las simulaciones temporales.
- `boolean`. Almacena el resultado de una operación lógica, que puede ser `true` o `false`.
- `severity_level`. Se usa en para dar avisos en simulación.
- `string`. Es una cadena de caracteres.
- `real`. Almacena un número en coma flotante.
- `bit`. Como su propio nombre indica, almacena un bit que puede tomar los valores 0 y 1. No ha de usarse en la práctica, siendo mejor usar `std_logic`.
- `bit_vector`. Es un vector de datos de tipo `bit`. Tampoco debe de usarse en la práctica, siendo recomendable usar `std_logic_vector` en su lugar.

Tipos de datos definidos por el usuario

Los tipos definidos por el usuario más usados en la práctica son `std_logic` y `std_logic_vector`, que como habrá supuesto no es más que un vector de `std_logic`. Ambos están definidos en el paquete `std_logic_1164` de la librería `ieee`. Estos tipos, al contrario de `bit` definen una lógica de 9 valores para describir el estado de las señales. ¡9 valores! estará pensando. ¡Pero si en digital sólo hay ceros y unos! Si, es cierto, pero no todos los ceros y los unos son iguales. Los nueve valores que define `std_logic` son:

- `'U'` Sin inicializar.
- `'X'` Valor desconocido. Es imposible calcular el valor de la señal.
- `'0'` El cero lógico.
- `'1'` El uno lógico.
- `'Z'` Alta impedancia.
- `'W'` Desconocido débil.
- `'L'` Cero débil.
- `'H'` Uno débil.
- `'-'` *don't care*.

Sin entrar en demasiados detalles (ya los iremos viendo a lo largo del texto), estos valores permiten simulaciones más acordes con la realidad. Por ejemplo, cuando arranca la simulación, el simulador pone todas las señales a `'U'`, de forma que si hacemos por ejemplo una OR entre una señal a `'0'` y una sin inicializar, la salida se pone también a `'U'`. Así, si metemos la pata nos daremos cuenta en seguida porque se nos llenará la pantalla del simulador de `UUUU`. Si en cambio hubiésemos usado señales de tipo `bit`, como sólo puede valer 0 o 1, el simulador no puede decirnos que estamos haciendo algo mal y nos daremos cuenta cuando sea demasiado tarde. Otro caso típico es cortocircuitar dos salidas. En este caso si en la simulación una de ellas está a uno y otra a cero, es imposible saber el valor de la señal, por lo que el simulador la pondrá a `'X'`.

Definición de tipos

Si lo necesitamos, podemos crear un tipo usando la siguiente sintaxis:

```
type t_nombre_tipo is (lista de valores);
```

Este tipo de datos se denominan datos enumerados porque se da una lista de los valores que puede tomar el tipo.

El nombre del tipo definido `t_nombre_tipo` ha de ser cualquier identificador válido de VHDL. Se recomienda precederlo de `t_` para distinguirlo de señales y bloques. Por ejemplo si queremos crear un tipo para almacenar valores lógicos podemos escribir:

```
type t_logico is (cierto, falso);
```

Una vez definido el tipo, para crear una señal del tipo nuevo basta con hacer:

```
signal sig_logica : t_logico;
```

También se pueden crear subtipos, que son variables de un tipo limitadas a un rango de valores. Por ejemplo podemos crear un subtipo para almacenar números negativos de la siguiente forma:

```
subtype negativo is integer range -2147483647 to -1;
```

En VHDL existen dos subtipos de enteros predefinidos: `natural` que son todos los números desde el 0 hasta el positivo más grande (2147483646) y `positive` que va desde el 1 hasta el positivo más grande.

4.5.1. Constantes

Las constantes permiten crear descripciones más legibles y fáciles de modificar. En VHDL la declaración de una constante es:

```
constant c_nombre_constante : tipo := valor inicial;
```

Por ejemplo:

```
constant c_n_bits : integer := 8;
```

Para distinguir las constantes de las señales, precederemos su nombre por `c_`.

4.5.2. Vectores

En VHDL, al igual que en todos los lenguajes, también se pueden crear vectores. Éstos son también tipos derivados, y al contrario que otros lenguajes como C, antes de crear un vector es necesario crear un tipo. La sintaxis básica es:

```
type t_nombre_tipo is array (inicio downto fin) of tipo_elemento;
```

Por ejemplo:

```
type t_byte is array (7 downto 0) of std_logic;
```

En estas definiciones también se pueden usar constantes, lo que hace más fácil cambiar los tamaños de las señales:

```
constant c_n_bits : integer := 8;
type t_byte is array (c_n_bits-1 downto 0) of std_logic;
```

Inicialización de vectores

La inicialización de vectores se puede realizar de varias maneras. La primera es listando los valores de todos los elementos. Por ejemplo si hemos creado una señal de tipo `byte`:

```
signal mi_byte : t_byte;
```

Para inicializarla se puede hacer:

```
mi_byte <= ('1','0','1','0','0','0','0','1');
```

También se pueden asignar valores a posiciones del vector indicando el índice. En este caso los índices que no se hayan especificado pueden inicializarse mediante la palabra clave **`others`**. Así, la variable `my_byte` puede inicializarse al mismo resultado haciendo:

```
mi_byte <= (7=>'1', 5=>'1',0=>'1', others=>'0');
```

Esto último viene muy bien cuando queremos inicializar un vector a cero:

```
mi_byte <= (others=>'0');
```

Si el vector a inicializar es de tipo `std_logic`, también se puede usar una cadena de caracteres que contenga sólo los 9 caracteres válidos (`'0'`, `'1'`, `'U'`, etc.) Por ejemplo, el byte anterior puede inicializarse:

```
mi_byte <= "10100001";
```

La única precaución con este tipo de inicialización es que el número de bits especificados en la cadena ha de ser igual al tamaño del vector.

Por último, si queremos acceder a un elemento del vector se especifica el índice entre paréntesis. Así, en el ejemplo anterior `mi_byte(7)` será igual a 1 y `mi_byte(1)` será igual a 0. Incluso se puede acceder a un rango de bits. Por ejemplo la señal `mi_byte(7 downto 4)` será igual a `1010`.

	Aritméticos	Lógicos		Relacionales
+	suma	and	>	mayor
-	resta	or	<	menor
*	multiplicación	not	=	igual
/	división	nand	/=	distinto
mod	módulo	nor	<=	mayor o igual
rem	resto	xor	>=	menor o igual
abs	valor absoluto	xnor		
**	exponenciación			

Cuadro 4.1: Operadores definidos en VHDL

4.5.3. Operadores

El lenguaje define una serie de operadores sobre los tipos predefinidos, los cuales se listan en el cuadro 4.1.

Los operadores aritméticos están predefinidos para los tipos `integer`, `real`[9] y `time`. Los operadores lógicos están definidos para los tipos `bit` y `std_logic`. Por último, los operadores relacionales están definidos para `integer`, `std_logic_vector` y `bit_vector`. El resultado de estos operadores es un valor de tipo `boolean` que sólo puede ser usado en expresiones condicionales, como veremos a continuación.

Por último, sobre los vectores se define el operador concatenación &. Así, la expresión `"01"& '1' & "010"` es equivalente a `"011010"`.

4.6. Sentencias concurrentes

En VHDL existen dos tipos de sentencias que permiten asignar valores a señales en función de una expresión lógica o de un valor. La primera es la sentencia **`when`**, que la hemos usado ya en el ejemplo del multiplexor. La segunda es la sentencia **`with`**, que nos será muy útil para implantar tablas de verdad.

4.6.1. Sentencia `when`

La sintaxis de esta sentencia es la siguiente:

```
signal <= expresion      when condicion        else
          otra_expresion when otra_condicion   else
          y_otra_mas     when y_otra_condicion else
          la_ultima;
```

El circuito generado asigna la expresión de la primera condición que sea cierta a la señal de salida. Así, si en el ejemplo tanto `condicion` como `otra_condicion` son ciertas, a **`signal`** se le asignará `expresion`.

[9] Excluyendo **`mod`** y **`rem`** pues no tienen sentido.

Lo mejor como siempre es ver un ejemplo. Una forma de expresar un comparador de dos bits es el siguiente:

```
1  -- Comparador para números de dos bits.
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity Comparador2Bits is
7
8    port (
9      a, b      : in  std_logic_vector(1 downto 0); -- Entradas
10     a_mayor_b : out std_logic;                    -- Salidas
11     a_menor_b : out std_logic;
12     a_igual_b : out std_logic);
13
14 end Comparador2Bits;
15
16 architecture behavioral of Comparador2Bits is
17
18 begin  -- behavioral
19
20   a_mayor_b <= '1' when a>b else
21                '0';
22   a_menor_b <= '1' when a<b else
23                '0';
24   a_igual_b <= '1' when a=b else
25                '0';
26
27 end behavioral;
```

Como puede apreciar se ha utilizado la sentencia **when** para poner a uno las salidas del comparador cuando sus respectivas comparaciones son ciertas y a cero en caso contrario.

4.6.2. Sentencia **with**

La sentencia **when** nos permite asignar múltiples expresiones a una señal en función de múltiples condiciones. En ocasiones sin embargo interesa asignar múltiples valores a una señal en función del valor de otra señal. Esto es muy útil para implantar tablas de verdad directamente en VHDL, sin necesidad de obtener las ecuaciones lógicas. La sintaxis de la sentencia es:

```
with expresion_seleccion select
  signal <= expresion      when valor,
```

$A_1A_0B_1B_0$	a_mayor_b	a_menor_b	a_igual_b
0 0 0 0	0	0	1
0 0 0 1	0	1	0
0 0 1 0	0	1	0
0 0 1 1	0	1	0
0 1 0 0	1	0	0
0 1 0 1	0	0	1
0 1 1 0	0	1	0
0 1 1 1	0	1	0
1 0 0 0	1	0	0
1 0 0 1	1	0	0
1 0 1 0	0	0	1
1 0 1 1	0	1	0
1 1 0 0	1	0	0
1 1 0 1	1	0	0
1 1 1 0	1	0	0
1 1 1 1	0	0	1

Cuadro 4.2: Tabla de verdad de un comparador de dos bits

```
otra_expresion when otro_valor,
otra_mas       when otro mas,
la_ultima      when others;
```

El funcionamiento de la sentencia es el siguiente: la `expresion_seleccion` se compara con los valores `valor`, `otro_valor`, etc. Si se encuentra uno que coincide, la expresión de al lado se asigna a **`signal`**. Si no se encuentra ninguno, se asigna la expresión del **`when others`**.

Para ilustrar el uso, vamos a especificar el mismo ejemplo del comparador, teniendo en cuenta que su tabla de verdad es la mostrada en el cuadro 4.2. El código en VHDL es el siguiente:

```
1  -- Comparador para números de dos bits. Implantado mediante
2  -- una tabla de verdad
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6
7  entity Comparador2BitsT is
8
9    port (
10     a, b      : in  std_logic_vector(1 downto 0); -- Entradas
11     a_mayor_b : out std_logic;                    -- Salidas
12     a_menor_b : out std_logic;
13     a_igual_b : out std_logic);
```

```
14
15 end Comparador2BitsT;
16
17 architecture behavioral of Comparador2BitsT is
18   -- Las salidas y entradas se agrupan en vectores para poder
19   -- tratarlas de forma conjunta en la tabla de verdad.
20   signal salidas  : std_logic_vector(2 downto 0);
21   signal entradas : std_logic_vector(3 downto 0);
22 begin  -- behavioral
23
24   entradas <= a & b;
25   a_mayor_b <= salidas(2);
26   a_menor_b <= salidas(1);
27   a_igual_b <= salidas(0);
28
29   with entradas select
30     salidas <=
31     "001" when "0000",
32     "010" when "0001",
33     "010" when "0010",
34     "010" when "0011",
35     "100" when "0100",
36     "001" when "0101",
37     "010" when "0110",
38     "010" when "0111",
39     "100" when "1000",
40     "100" when "1001",
41     "001" when "1010",
42     "010" when "1011",
43     "100" when "1100",
44     "100" when "1101",
45     "100" when "1110",
46     "001" when "1111",
47     "000" when others;
48
49 end behavioral;
```

Para facilitar la escritura de la tabla se han agrupado las entradas en un solo vector usando el operador &, tal como puede apreciar en la línea 24. Una vez hecho esto, son la sentencia **with** se asigna un valor a la señal `salidas` en función del valor de las entradas según la tabla de verdad. En las líneas 25 a 27 este vector salidas se asigna a cada una de las salidas individuales del comparador.

4.7. Ejercicios

1. Realice una descripción puramente estructural del circuito para comparar dos números de dos bits mostrado en la sección 4.4.1. Para ello ha de crear el bloque PuertaOR e instanciarlo en la arquitectura del bloque CircuitoAlarma2.
2. Describa en VHDL los circuitos diseñados en los ejercicios 6 y 7 de la página 34 del capítulo 2.

CAPÍTULO 5

Circuitos Aritméticos

En este capítulo se estudian una serie de circuitos para realizar operaciones aritméticas con números binarios. Se estudia en primer lugar el sumador. A continuación veremos cómo implantar un restador usando el complemento a dos. Por último se estudiará cómo construir un multiplicador.

5.1. Sumador de un bit

El estudio del sumador se va a dividir en dos partes. En primer lugar se estudia un sumador de un bit sin tener en cuenta el acarreo de entrada, al que se le denomina semisumador. A continuación se estudiará como ampliar este semisumador para tener en cuenta el acarreo de entrada y construir así el denominado sumador completo.

5.1.1. Semisumador

En el cuadro 5.1(a) se muestra la tabla de verdad del semisumador. Si obtenemos los minitérminos para las salidas de acarreo c y de suma s:

$$\begin{aligned} c &= a \cdot b \\ s &= \overline{a} \cdot b + a \cdot \overline{b} = a \oplus b \end{aligned}$$

La implantación del circuito se muestra en la figura 5.1.

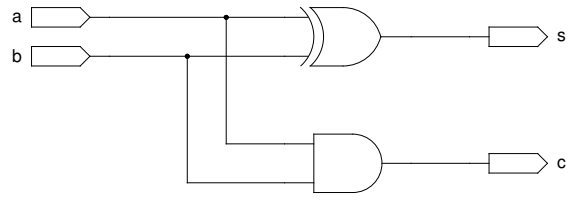


Figura 5.1: Circuito Semisumador.

a	+	b	c	s
0		0	0	0
0		1	0	1
1		0	0	1
1		1	1	0

(a) Tabla de sumar sin acarreo de entrada

c_i	+	a_i	+	b_i	c_{i+1}	s_i
0		0		0	0	0
0		0		1	0	1
0		1		0	0	1
0		1		1	1	0
1		0		0	0	1
1		0		1	1	0
1		1		0	1	0
1		1		1	1	1

(b) Tabla de sumar con acarreo de entrada

Cuadro 5.1: Tablas de suma

5.1.2. Sumador completo

El semisumador es un circuito muy simple, pero no es muy útil que se diga, porque sólo sirve para sumar dos números de un bit. Como en la práctica los números son de más bits, es necesario diseñar un sumador que tenga en cuenta el acarreo de entrada. Para ello, a partir de la tabla 5.1(b) podemos obtener la suma de minitérminos de las salidas de la suma del bit número i, s_i y c_{i+1}, y simplificarlas mediante manipulaciones algebraicas. Si comenzamos por la suma s_i:

$$s_i = \overline{c_i} \cdot \overline{a_i} \cdot b_i + \overline{c_i} \cdot a_i \cdot \overline{b_i} + c_i \cdot \overline{a_i} \cdot \overline{b_i} + c_i \cdot a_i \cdot b_i$$

Si sacamos factor común en los dos primeros minitérminos a $\overline{c_i}$ y en los dos últimos a c_i, teniendo en cuenta que $\overline{a_i} \cdot b_i + \cdot a_i \cdot \overline{b_i} = a \oplus b$, y que $\overline{a_i} \cdot \overline{b_i} + \cdot a_i \cdot b_i = \overline{a_i \oplus b_i}$ se obtiene:

$$s_i = \overline{c_i} \cdot (a \oplus b) + c_i \cdot \overline{(a_i \oplus b_i)}$$

Que nuevamente se puede simplificar usando una xor a:

$$s_i = c_i \oplus (a \oplus b) \tag{5.1}$$

La ecuación del acarreo de salida se obtiene de igual forma. Partiendo de la suma de minitérminos:

$$c_{i+1} = \overline{c_i} \cdot a_i \cdot b_i + c_i \cdot \overline{a_i} \cdot b_i + c_i \cdot a_i \cdot \overline{b_i} + c_i \cdot a_i \cdot b_i$$

Sacando factor común a $a_i \cdot b_i$ en el primer y último término y a c_i en los términos segundo y tercero se obtiene:

$$c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i \tag{5.2}$$

Este circuito se puede construir a partir de dos semisumadores y una puerta OR, tal como se muestra en la izquierda de la figura 5.2. El primer semisumador calcula la suma de $a_i + b_i$ y el segundo le suma al resultado anterior el bit c_i. En la derecha de la figura se muestra una representación del bloque, ya que lo usaremos para construir sumadores de n bits en la siguiente sección.

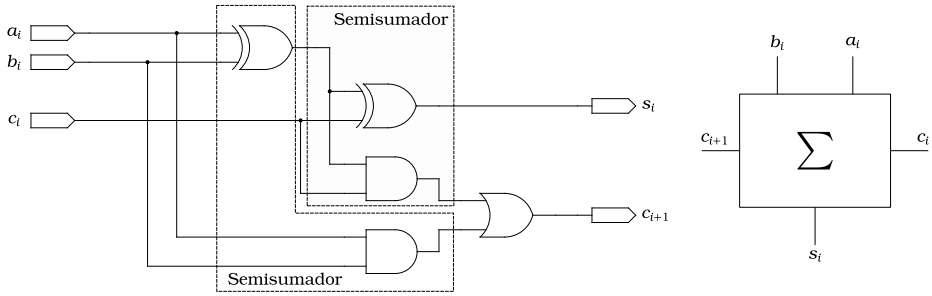


Figura 5.2: Circuito sumador completo.

5.2. Sumador de palabras de n bits

Para realizar la suma de dos palabras de n bits basta con usar n sumadores completos como el estudiado en la sección anterior. Existen dos métodos para gestionar los acarreos: acarreo serie, que es fácil de construir pero lento y con predicción de acarreo, que es más rápido pero necesita más lógica.

5.2.1. Sumador de n bits con acarreo serie

Los sumadores con acarreo serie son los más fáciles de construir y de entender. Estos sumadores gestionan los acarreos de la misma forma que lo hacemos nosotros cuando realizamos la suma con lápiz y papel. El acarreo correspondiente al bit menos significativo es cero y el resto de bits toman como acarreo de entrada el acarreo salida de la etapa anterior. El circuito resultante es el mostrado en la figura 5.3, donde se muestra un sumador para números de 4 bits. Tenga en cuenta que el acarreo de salida c_4, aunque parezca inútil, sirve para averiguar si se ha producido un desbordamiento en la suma, tal como se estudió en el capítulo 3.

Nótese que ampliar este sumador a un número mayor de bits es bien simple: basta con añadir tantos sumadores como bits nos hagan falta y conectar los acarreos

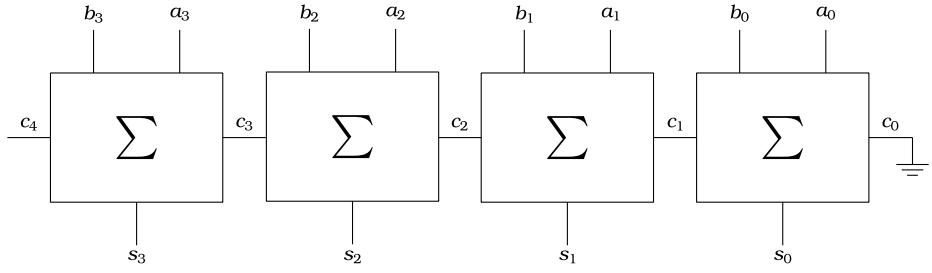


Figura 5.3: Circuito sumador de n bits con acarreo serie.

en serie.

Retardo en la salida

A pesar de su gran sencillez, el sumador con acarreo serie presenta un grave inconveniente: es muy lento, ya que la suma sólo será válida cuando el acarreo se haya propagado por todas las etapas. Por ejemplo si realiza la suma:

$$\begin{array}{rr} & {\scriptstyle 1\ 1\ 1} \\ & 1\ 1\ 1\ 1 \\ + & 0\ 0\ 0\ 1 \\ \hline & 1\,0\,0\,0\,0 \end{array}$$

notará que el acarreo generado al sumar los dos bits menos significativos ha de propagarse por el resto de los bits hasta llegar al final. En general, para un sumador de n etapas, si suponemos que todos los bits de los datos a y b llegan a la vez a las entradas, el retardo en realizar la suma será:

$$t_{suma} = t_{a_0 b_0 \to c_1} + (n-1) \cdot t_{c_i \to c_{i+i}}$$

En donde $t_{a_0 b_0 \to c_1}$ es el retardo desde las entradas del sumador para los bits menos significativos hasta la salida de acarreo de dicho sumador y $t_{c_i \to c_{i+i}}$ es el retardo entre la entrada de acarreo de un sumador intermedio hasta su salida de acarreo.

Descripción en VHDL

La descripción de un sumador de n bits en VHDL puede hacerse de forma estructural usando en primer lugar un bloque para implementar el sumador completo de un bit y a continuación crear un bloque que instancie n sumadores de un bit conectando los acarreos en serie. La descripción del sumador de un bit se realiza a partir de las ecuaciones (5.1) y (5.2).

```
1  -- Sumador completo de un bit con salida de acarreo. Se usa
2  -- para construir un sumador con acarreo serie
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6
7  entity Sumador1Bit is
8
9    port (
10     a_i, b_i  : in  std_logic;          -- Entradas de datos
11     c_i       : in  std_logic;          -- Acarreo de entrada
12     s_i       : out std_logic;          -- Salida
13     c_i_mas_1 : out std_logic);         -- Acarreo de salida
14
```

```
15 end Sumador1Bit;
16
17 architecture behavioral of Sumador1Bit is
18
19 begin  -- behavioral
20
21   s_i       <= c_i xor (a_i xor b_i);
22   c_i_mas_1 <= (a_i and b_i) or ((a_i xor b_i) and c_i);
23
24 end behavioral;
```

Lo único que hay que destacar del código ha sido la necesidad de añadir paréntesis en la línea 22, ya que la precedencia de los operadores lógicos en VHDL para el tipo `std_logic` no está definida.

El sumador de 4 bits se crea instanciando cuatro sumadores de 1 bit y conectando sus acarreos en serie, tal como se ha mostrado en la figura 5.3. Además el acarreo del bit cero se conecta a `'0'` y el acarreo de la última etapa se deja como salida para poder detectar el desbordamiento en la suma de números sin signo.

La sentencia `for ... generate`

Igual está temblando ante la idea de tener que instanciar cuatro componentes directamente en VHDL, pero no se preocupe, que afortunadamente no vamos a tener que teclear tanto, sobre todo cuando tengamos que crear un sumador de 32 bits. Por suerte los diseñadores del lenguaje pensaron en una manera de instanciar varias veces el mismo componente mediante la sentencia **`for ... generate`**. Por tanto antes de estudiar cómo implantar el sumador vamos a ver la sintaxis de dicha sentencia:

```
[etiqueta:] for var_bucle in v_inicial to v_final generate

end generate [etiqueta];
```

Como puede apreciar la sentencia consta de una etiqueta opcional seguida de la palabra clave **`for`**. A continuación se necesita especificar una variable que se usará para controlar la "iteración" del bucle. Esta variable es de tipo entero y se crea automáticamente al entrar en el bucle, por lo que no tiene que preocuparse de declararla antes. Después de la palabra clave **`in`** se especifican los valores inicial y final de la variable de control separados por la palabra clave **`to`**[1]. Por último la palabra clave **`generate`** da comienzo al cuerpo del bucle, el cual termina con **`end generate`**.

Tenga en cuenta que esta sentencia, al contrario que un bucle en *software* no repite la ejecución de instrucciones, sino que en cada "iteración" del bucle crea un

[1] Si se necesita una variable decreciente se puede usar un rango del tipo `3 downto 0`.

componente particularizado para el valor actual de la variable de control, tal como veremos a continuación.

Sumador de 4 bits en VHDL

La descripción en VHDL de un sumador de 4 bits que vamos a realizar utiliza un bucle **for ... generate** para instanciar cuatro componentes Sumador1Bit.

```
1  -- Sumador con acarreo serie para números de 4 bits. Se usa
2  -- una descripción estructural instanciando 4 sumadores de
3  -- 1 bit.
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7
8  entity Sumador4Bits is
9
10   port (
11     a, b  : in  std_logic_vector(3 downto 0);  -- Entradas
12     s     : out std_logic_vector(3 downto 0);  -- Salida
13     c_out : out std_logic);           -- Acarreo de salida
14
15 end Sumador4Bits;
16
17 architecture structural of Sumador4Bits is
18   signal c : std_logic_vector(4 downto 0);  -- Acarreos
19                                             -- intermedios
20   component Sumador1Bit
21     port (
22       a_i, b_i  : in  std_logic;
23       c_i       : in  std_logic;
24       s_i       : out std_logic;
25       c_i_mas_1 : out std_logic);
26   end component;
27 begin  -- structural
28
29   c(0) <= '0';                -- El acarreo inicial es cero
30   c_out <= c(4);              -- y el final c(4)
31
32   GenSum : for i in 0 to 3 generate
33     -- instance "Sumador1Bit_1"
34     Sumador1Bit_1 : Sumador1Bit
35       port map (
36         a_i       => a(i),
37         b_i       => b(i),
```

```
38          c_i        => c(i),
39          s_i        => s(i),
40          c_i_mas_1  => c(i+1));
41    end generate GenSum;
42
43  end structural;
```

En primer lugar conviene destacar que en la línea 18 se han creado cuatro señales para los cuatro acarreos que necesita el circuito. A continuación se declara el componente y se pasa a describir la arquitectura. En ésta, en primer lugar (líneas 29 y 30) se pone el acarreo inicial `c(0)` a cero y se conecta el final `c(4)` a la salida de acarreo `c_out`.

Para instanciar los cuatro sumadores se ha usado un bucle **`for ... generate`** en el que se usa `i` como variable de control y se la hace variar de 0 a 3. Para cada valor de la variable, se genera un sumador de un bit al que se conectan las señales `a(i)`, `b(i)`, etc. Lo único destacable es que en la línea 40 se ha conectado la salida de acarreo del sumador a la señal de acarreo de entrada del siguiente, que no es otra que `c(i+1)`. Así, en la primera iteración del bucle (`i=0`), el componente que se instancia sería:

```
Sumador1Bit_1 : Sumador1Bit
  port map (
    a_i        => a(0),
    b_i        => b(0),
    c_i        => c(0),
    s_i        => s(0),
    c_i_mas_1  => c(1));
```

Por si no se cree que esto funciona, en la figura 5.4 se muestra en circuito generado por el sintetizador del entorno Quartus II. El retardo obtenido en la simulación ha sido de 14,002 ns.

5.2.2. *Sumador de n bits con predicción de acarreo*

Para evitar el excesivo retardo de la propagación del acarreo en el sumador anterior existe una alternativa que permite calcular los acarreos de todos los sumadores

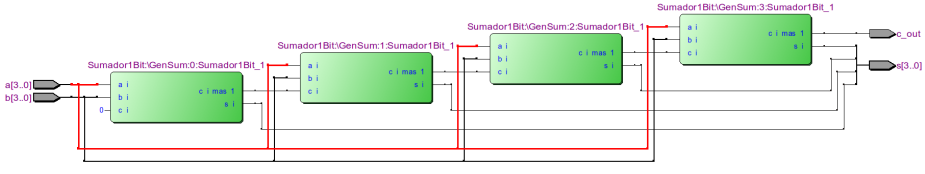


Figura 5.4: Circuito sumador de 4 bits generado a partir de la descripción anterior.

de un bit en paralelo. Para realizar este cálculo recordemos que la ecuación del acarreo obtenida en la sección 5.1.2 es:

$$c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i$$

Si nos fijamos en la ecuación, vemos que cuando el término $a_i \cdot b_i$ es 1 se produce siempre un acarreo de salida. Decimos entonces que $a_i \cdot b_i$ genera un acarreo en la etapa i y a esta expresión la denominamos g_i. Por otro lado, el término $a_i \oplus b_i$ produce un acarreo en la salida sólo si c_i es uno, es decir, si hay un acarreo en la entrada. Decimos entonces que el término $a_i \oplus b_i$ propaga el acarreo en la etapa i y lo denominamos p_i. Teniendo esto en cuenta, la ecuación anterior la podemos escribir como:

$$c_{i+1} = g_i + p_i \cdot c_i$$

A partir de esta ecuación el cálculo de los acarreos para cada bit es sencillo. Si por ejemplo tenemos un sumador de cuatro bits, las ecuaciones de los acarreos de cada etapa serán:

$$c_1 = g_0 + p_0 \cdot c_0 \tag{5.3}$$
$$c_2 = g_1 + p_1 \cdot c_1 \tag{5.4}$$
$$c_3 = g_2 + p_2 \cdot c_2 \tag{5.5}$$
$$c_4 = g_3 + p_3 \cdot c_3 \tag{5.6}$$

Hasta ahora no hemos conseguido nada, porque en estas ecuaciones se sigue usando el acarreo de la etapa anterior para el cálculo de la siguiente. No obstante, si sustituimos la ecuación (5.3) en la ecuación (5.4) obtenemos:

$$c_2 = g_1 + p_1 \cdot (g_0 + p_0 \cdot c_0) = g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \tag{5.7}$$

Esta ecuación puede leerse como: "habrá acarreo hacia la etapa 2 si o bien en la etapa 1 se ha generado un acarreo (g_1) o si la etapa 1 ha propagado un acarreo y la etapa 0 lo ha generado ($p_1 \cdot g_0$) o si las etapas 1 y 0 lo han propagado y había un acarreo a la entrada de la etapa 0 ($p_1 \cdot p_0 \cdot c_0$)."

Repitiendo el proceso para la etapa siguiente, si sustituimos la expresión de c_2 que acabamos de obtener en la ecuación (5.5) obtenemos:

$$c_3 = g_2 + p_2 \cdot c_2 = g_2 + p_2 \cdot (g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0) = g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \tag{5.8}$$

Por si no se ha dado cuenta, aparece un patrón común en las ecuaciones, que seguro que si obtenemos la siguiente lo ve más claro. Repitiendo el proceso para la siguiente etapa se obtiene:

$$c_4 = g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0 \tag{5.9}$$

¿No lo ha visto aún? A ver si poniéndolas todas juntas. . .

$$
\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\
c_3 &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\
c_4 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
\end{aligned}
$$

Ahora sí, ¿verdad? Bromas aparte, la ecuación para la siguiente etapa es fácil de escribir, pero ya empieza a ser difícil de implantar y veremos en seguida cómo hacer sumadores con predicción de acarreo de más de 4 bits. El otro aspecto que hay que recalcar de estas ecuaciones es que todas se forman con sólo dos niveles de puertas AND–OR, por lo que el retardo para calcular todos los acarreos será mucho menor que antes.

La estructura del sumador es la mostrada en la figura 5.5. Como puede observar, ahora la unidad de predicción de acarreo (*carry look ahead*) se encarga de calcular los acarreos de entrada de cada uno de los sumadores de un bit, aunque para ello necesita un poco de ayuda de éstos sumadores en forma de las señales g_i y p_i. Estas señales, si observa la figura ya estaban presentes en el sumador completo mostrado en la figura 5.2, por lo que no necesitan ser calculadas de nuevo.

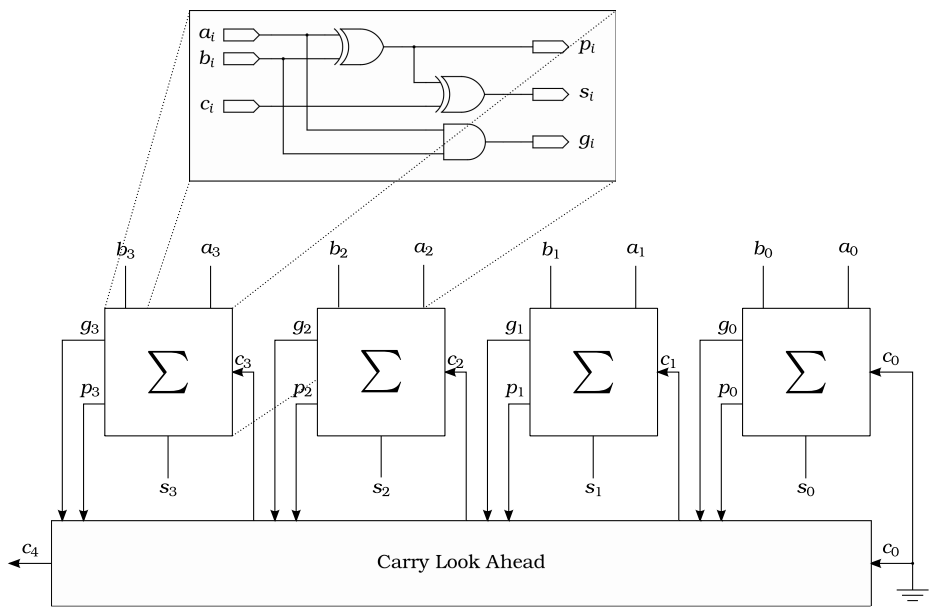


Figura 5.5: Circuito sumador de n bits con predicción de acarreo.

Descripción en VHDL

La descripción del sumador con predicción de acarreo es simular a la realizada para el sumador con acarreo serie. En primer lugar se describe el sumador de un bit, que es muy similar al usado para el sumador anterior, con la salvedad de que en lugar de generar un acarreo de salida c_{i+1} calcula las señales de generación y propagación de acarreo g_i y p_i.

```
1  -- Sumador completo de un bit con salidas de predicción
2  -- y generación de acarreo. Se usa para construir un
3  -- sumador con predicción de acarreo.
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7
8  entity Sumador1BitPG is
9
10   port (
11     a_i, b_i : in  std_logic;       -- Entradas de datos
12     c_i      : in  std_logic;       -- Acarreo de entrada
13     s_i      : out std_logic;       -- Salida
14     p_i, g_i : out std_logic);      -- Pred y Gen de Acarreo
15
16 end Sumador1BitPG;
17
18 architecture behavioral of Sumador1BitPG is
19
20 begin  -- behavioral
21
22   s_i <= c_i xor (a_i xor b_i);
23   p_i <= a_i xor b_i;
24   g_i <= a_i and b_i;
25
26 end behavioral;
```

Nótese que en la línea 23 aparece repetido el término `a_i xor b_i`. Esto no genera un circuito ineficiente porque el sintetizador se encarga de detectar este tipo de situaciones y genera la lógica adecuada, tal como se demuestra en la figura 5.6.

Si no se queda tranquilo se puede declarar una variable para almacenar este valor intermedio, ya que en VHDL no se puede leer el valor de una salida. En este caso el código de la arquitectura quedaría:

```
architecture behavioral of Sumador1BitPG is
  signal p_i_int: std_logic;
begin  -- behavioral
```

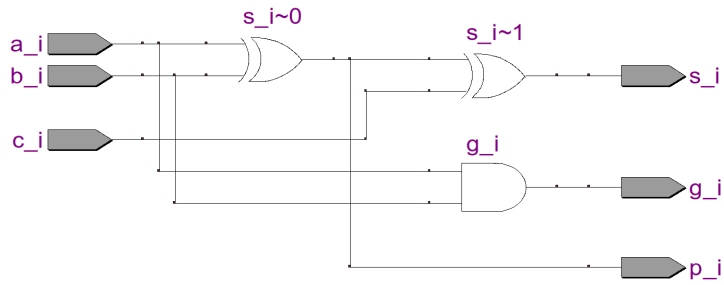


Figura 5.6: Circuito sumador de un bit con salidas p_i y g_i.

```
  p_i_int <= a_i xor b_i;
  s_i <= c_i xor p_i_int;
  p_i <= p_i_int;
  g_i <= a_i and b_i;

end behavioral;
```

Donde como puede apreciar se ha creado la señal `p_i_int` para almacenar el valor de la salida p_i. El circuito generado por el sintetizador es en ambos casos el mismo.

La unidad de predicción de acarreo es también fácil de describir: basta con escribir las ecuaciones obtenidas en la sección anterior:

```
1  -- Unidad de predicción de acarreo para un sumador de 4 bits
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity CarryLookAhead is
7
8    port (
9      -- Entradas de propagación y generación
10     g, p : in  std_logic_vector(3 downto 0);
11     c_0  : in  std_logic;          -- Acarreo de entrada
12     -- Acarreos de salida
13     c    : out std_logic_vector(4 downto 1));
14
15 end CarryLookAhead;
16
17 architecture behavioral of CarryLookAhead is
18
19 begin  -- behavioral
20
```

```
21   c(1) <= g(0) or (p(0) and c_0);
22   c(2) <= g(1) or (p(1) and g(0)) or (p(1) and p(0) and c_0);
23   c(3) <= g(2) or (p(2) and g(1)) or (p(2) and p(1) and g(0))
24           or (p(2) and p(1) and p(0) and c_0);
25   c(4) <= g(3) or (p(3) and g(2)) or (p(3) and p(2) and g(1))
26           or (p(3) and p(2) and p(1) and g(0))
27           or (p(3) and p(2) and p(1) and p(0) and c_0);
28
29 end behavioral;
```

Por último, el sumador de cuatro bits es muy parecido al anterior. Lo único que cambia ahora es la presencia de las señales de generación y propagación y la instanciación del componente CarryLookAhead para la predicción de los acarreos.

```
1  -- Sumador con predicción de acarreo para números de 4 bits.
2  -- Se usa una descripción estructural instanciando 4 suma-
3  -- dores de 1 bit y una unidad de predicción de acarreo.
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7
8  entity Sumador4BitsPA is
9
10   port (
11     a, b  : in  std_logic_vector(3 downto 0);  -- Entradas
12     s     : out std_logic_vector(3 downto 0);  -- Salida
13     c_out : out std_logic);           -- Acarreo de salida
14
15 end Sumador4BitsPA;
16
17 architecture structural of Sumador4BitsPA is
18   -- Acarreos intermedios
19   signal c : std_logic_vector(4 downto 0);
20   -- Generación y Propagación
21   signal g, p : std_logic_vector(3 downto 0);
22
23   component Sumador1BitPG
24     port (
25       a_i, b_i : in  std_logic;
26       c_i      : in  std_logic;
27       s_i      : out std_logic;
28       p_i, g_i : out std_logic);
29   end component;
30
31   component CarryLookAhead
```

```
32      port (
33        g, p : in  std_logic_vector(3 downto 0);
34        c_0  : in  std_logic;
35        c    : out std_logic_vector(4 downto 1));
36    end component;
37
38  begin  -- structural
39
40    c(0) <= '0';                -- El acarreo inicial es cero
41    c_out <= c(4);              -- y el final c(4)
42
43    GenSum : for i in 0 to 3 generate
44    -- instance "Sumador1BitPG_1"
45    Sumador1BitPG_1: Sumador1BitPG
46      port map (
47        a_i => a(i),
48        b_i => b(i),
49        c_i => c(i),
50        s_i => s(i),
51        p_i => p(i),
52        g_i => g(i));
53
54    end generate GenSum;
55
56    -- instance "CarryLookAhead_1"
57    CarryLookAhead_1: CarryLookAhead
58      port map (
59        g   => g,
60        p   => p,
61        c_0 => c(0),
62        c   => c(4 downto 1));
63
64  end structural;
```

5.2.3. *Componentes genéricos en VHDL*

En la sección anterior se ha creado un sumador de 4 bits de una forma muy cómoda en VHDL. Cambiar el sumador anterior a uno con un número distinto de bits es también más o menos fácil, pues basta con cambiar los índices de los vectores y los bucles. El problema es que esto es, además de un rollo, muy propenso a errores. Afortunadamente los diseñadores del lenguaje VHDL pensaron en ello y permitieron que los componentes puedan tener parámetros que permiten configurar ciertas constantes internas al componente. Al instanciar el componente podemos

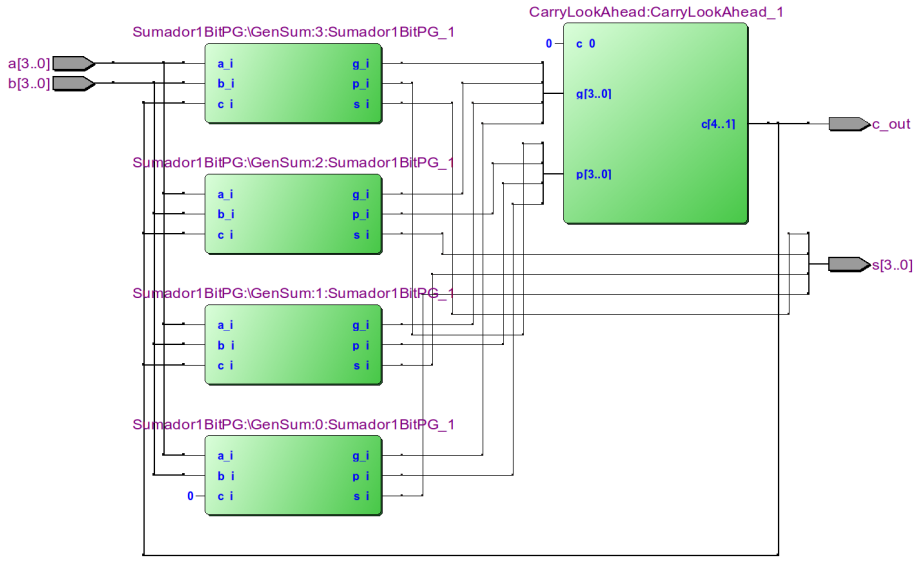


Figura 5.7: Circuito sumador de 4 bits con propagación de acarreo generado por Quartus II.

darle el valor que necesitemos, creando así un componente adaptado a cada momento. Para ilustrar el uso de esta funcionalidad del VHDL vamos a diseñar un sumador con propagación de acarreo con un número genérico de bits.

La sentencia generic

Al definir el interfaz de un componente se puede incluir además de las definiciones de puertos, las definiciones de parámetros (**generic**). En la sección 4.3.1 se mostró la sintaxis para definir la interfaz del bloque, aunque no se entró en detalle sobre los genéricos. La sintaxis incluyendo dichos genéricos es:

```
entity NombreDeEntidad is
  [ generic (
        parametro_1 : tipo:= valor_por_defecto;
        parametro_2 : tipo:= valor_por_defecto;

        parametro_n : tipo:= valor_por_defecto); ]

  port (nombres de señales : modo tipo;
        nombres de señales : modo tipo;
```

```
        nombres de señales : modo tipo);
end [NombreDeEntidad];
```

Como se puede observar, las declaraciones de parámetros tienen un formato parecido al de los puertos. La única diferencia es que no hay modo de entrada/salida, pues no tiene sentido y hay un valor por defecto, que es el que se usa si no se especifica ningún valor para el parámetro al instanciar el componente.

Por último destacar que para distinguir los parámetros de las variables y constantes, precederemos siempre su nombre por `g_`.

Sumador de n bits con acarreo serie

Para implantar un sumador de n bits con acarreo serie usaremos un parámetro, al que denominaremos `g_data_w`. Por tanto, para convertir el sumador de 4 bits descrito anteriormente a un componente genérico basta con sustituir los índices de los vectores y bucles.

```
1  -- Sumador con acarreo serie para números de N bits. Se usa
2  -- una descripción estructural instanciando N sumadores de
3  -- 1 bit.
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7
8  entity SumadorNBits is
9    generic (
10     g_data_w : integer := 4);    -- Nº bits de los datos
11   port (
12     a, b  : in  std_logic_vector(g_data_w-1 downto 0);
13     s     : out std_logic_vector(g_data_w-1 downto 0);
14     c_out : out std_logic);           -- Acarreo de salida
15
16 end SumadorNBits;
17
18 architecture structural of SumadorNBits is
19   -- Acarreos intermedios
20   signal c : std_logic_vector(g_data_w downto 0);
21
22   component Sumador1Bit
23     port (
24       a_i, b_i  : in  std_logic;
25       c_i       : in  std_logic;
26       s_i       : out std_logic;
27       c_i_mas_1 : out std_logic);
28   end component;
```

```
29
30 begin  -- structural
31
32   c(0)  <= '0';          -- El acarreo inicial es cero
33   c_out <= c(g_data_w); -- y el final c(g_data_w)
34
35   GenSum : for i in 0 to g_data_w-1 generate
36   -- instance "Sumador1BitPG_1"
37   Sumador1Bit_1: Sumador1Bit
38     port map (
39       a_i => a(i),
40       b_i => b(i),
41       c_i => c(i),
42       s_i => s(i),
43       c_i_mas_1 => c(i+1));
44
45   end generate GenSum;
46
47 end structural;
```

En primer lugar destacar que el genérico creado se ha inicializado al valor por defecto de 4. Así, si al instanciar el componente no decimos nada, se creará el mismo componente descrito en la sección anterior. Por otro lado, si compara este sumador con el de 4 bits, verá que lo único que hemos hecho es sustituir el 3 de los índices de los vectores y el rango del **for** por `g_data_w-1` y el 4 de la salida del acarreo (línea 33) por `g_data_w`. El resto de la descripción es exactamente igual.

Para instanciar el componente, lo único que hay que hacer es incluir una sentencia **generic map** en la que se especifica el valor que queremos darle al genérico. Así, para instanciar un sumador de 8 bits bastaría con declararlo e instanciarlo, tal como se muestra a continuación:

```
architecture structural of MiCircuito is

  component SumadorNBits
    generic (
      g_data_w : integer);
    port (
      a, b  : in  std_logic_vector(g_data_w-1 downto 0);
      s     : out std_logic_vector(g_data_w-1 downto 0);
      c_out : out std_logic);
  end component;

begin  -- structural

  -- instance "SumadorNBits_1"
```

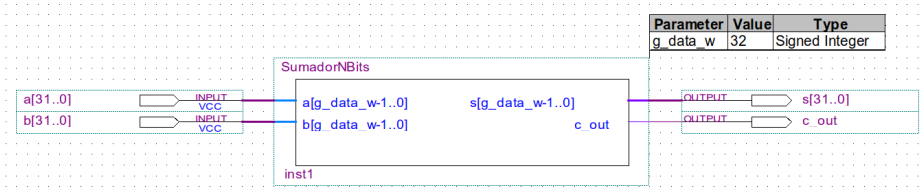
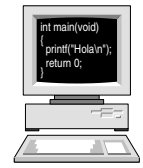


Figura 5.8: Sumador de 32 bits usando un componente genérico con Quartus II.

```
  SumadorNBits_1: SumadorNBits
    generic map (
      g_data_w => 7)
    port map (
      a     => a,
      b     => b,
      s     => s,
      c_out => c_out);

end structural;
```



Realice el ejercicio 1

Si en lugar de instanciarlo en una descripción VHDL desea incluir el componente en un esquema, los entornos CAD permiten también instanciar estos componentes y nos dejan cambiar fácilmente el parámetro. Por ejemplo, en la figura 5.8 se muestra un ejemplo en Quartus II en el que se ha creado un sumador de 32 bits usando este componente genérico.

5.2.4. *Descripción de sumadores en VHDL en alto nivel*

Dado que el sumador es un componente fundamental en los sistemas digitales, los sintetizadores de VHDL permiten generar sumadores mucho más fácilmente de lo que hemos visto en esta sección. Por ello, aunque el VHDL nativo sólo soporta la suma para los tipos enteros y reales, se ha desarrollado el paquete `ieee.numeric_std` que define los tipos `signed` y `unsigned` sobre los que si es posible usar el operador de suma. Ambos tipos son también vectores cuyos elementos son `std_logic`, pero con la ventaja de que el operador suma está definido para ellos. Además de estos tipos se definen en el mismo paquete funciones para convertir entre `std_logic_vector` y los tipos `signed` y `unsigned`. También se pueden definir señales (o puertos) de este tipo siendo la sintaxis la misma que para un `std_logic_vector`:

```
  signal us : unsigned(7 downto 0);
  signal s  : unsigned(7 downto 0);
```

No obstante se recomienda usar como señales `std_logic_vector` y realizar las conversiones cuando sea necesario, tal como veremos en esta sección.

Funciones de conversión

Para convertir una señal del tipo `std_logic_vector` a `signed` o `unsigned` y viceversa existen las funciones:

- `unsigned(un_std_logic_vector)`. Convierte una señal `std_logic_vector` al tipo `unsigned`. El número de bits del resultado es el mismo que el de la señal original.
- `signed(un_std_logic_vector)`. Convierte una señal `std_logic_vector` al tipo `signed`. El número de bits del resultado es el mismo que el de la señal original.
- `std_logic_vector(un_signed_o_unsigned)`. Convierte una señal `signed` o `unsigned` a `std_logic_vector`. El número de bits del resultado es el mismo que el de la señal original.

El operador + con los tipos `signed` y `unsigned`

VHDL permite la llamada sobrecarga de operadores, lo cual consiste en que el compilador ante un operador como por ejemplo + selecciona el algoritmo correcto en función del tipo de datos que se usan en la operación. Así, si el compilador se encuentra con la expresión `ent_1 + ent_2`, en donde ambas variables son de tipo entero, usa la operación de suma para números enteros; pero si se encuentra con `unsig_1 + unsig_2` en la que ambos operadores son de tipo `unsigned`, usa la operación de suma para el tipo `unsigned`.

La única característica resaltable de esta operación es que el resultado será un dato cuyo número de bits es igual al número de bits de los datos, si ambos tienen el mismo número de bits. Si por el contrario los datos tienen distinto número de bits, el resultado tendrá el mismo tamaño que el dato con mayor número de bits. Así, si por ejemplo sumamos dos números de 8 bits, el resultado también tendrá 8 bits. Si en cambio sumamos un número de 8 bits con otro de 4, el resultado tendrá 8 bits.

Lo anterior significa que si hacemos la siguiente suma:

```
  signal a, b, r: unsigned(7 downto 0);
...
  r <= a+b;
```

perdemos el acarreo de salida, con lo cual no podemos detectar si ha ocurrido un desbordamiento. No obstante podemos hacer un pequeño truco: extender los datos en un bit para que la suma se haga con un bit adicional y extraer luego ese bit del resultado. En ese caso el código quedaría:

```
  signal a, b, r : unsigned(7 downto 0);
  signal r_int   : unsigned(8 downto 0);
  signal c_out   : std_logic;
...
  r_int <= unsigned('0'&a) + unsigned('0'&b);
  r     <= r_int(7 downto 0);
```

```
  c_out <= r_int(8);
```

Nótese que ha sido necesario convertir de nuevo a unsigned el dato '0'&a debido a que al concatenar un std_logic a un unsigned el compilador por prudencia ya no supone que el vector resultante sigue siendo un número sin signo y lo interpreta como un std_logic_Vector. Son los "incordios" de que el lenguaje sea de "tipado" fuerte; pero recuerde que en el fondo todo esto es para evitar en lo posible errores.

Sumador de 4 bits usando el operador +

Aplicando todo lo anterior, el proceso a seguir para implantar una suma usando el paquete numeric_std consta de los siguientes pasos:

1. Obviamente el primer paso es añadir en la declaración de librerías y paquetes la línea **use** ieee.numeric_std.**all** para incluir este paquete.
2. Convertir los datos de std_logic_vector a signed o unsigned según el tipo de dato que esté almacenado en el vector.
3. Hacer la suma.
4. Convertir el resultado de nuevo a std_logic_vector

En el siguiente ejemplo se vuelve a describir un sumador de 4 bits usando el proceso descrito:

```
1  -- Sumador para números de 4 bits.
2  -- Se usa una descripción en alto nivel usando el
3  -- operador suma con tipos unsigned.
4
5  library ieee;
6  use ieee.std_logic_1164.all;
7  use ieee.numeric_std.all;
8
9  entity Sumador4BitsAN is
10   port (
11     a, b  : in  std_logic_vector(3 downto 0);  -- Entradas
12     s     : out std_logic_vector(3 downto 0);  -- Salida
13     c_out : out std_logic);           -- Acarreo de salida
14
15 end Sumador4BitsAN;
16
17 architecture behavioral of Sumador4BitsAN is
18   signal s_int : unsigned(4 downto 0);
19 begin  -- behavioral
20
21   s_int <= unsigned('0'&a)+unsigned('0'&b);
22   s     <= std_logic_vector(s_int(3 downto 0));
23   c_out <= s_int(4);
```

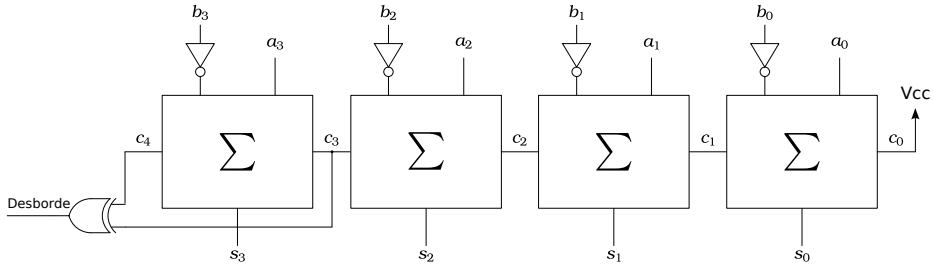
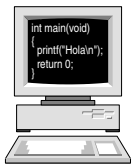


Figura 5.9: Restador de 4 bits.

```
24
25 end behavioral;
```

En primer lugar destacar que se ha creado una señal de tipo `unsigned` para almacenar el resultado de la suma en 5 bits y así poder obtener el acarreo. A continuación se ha procedido a añadir un cero como bit más significativo a los datos de entrada `a` y `b` y a convertirlos al tipo `unsigned` antes de sumarlos. Con ello se realiza la suma con 5 bits. Una vez realizada la suma, se procede a copiar los 4 bits menos significativos a la salida, convirtiéndolos previamente a `std_logic_vector` para que el compilador no se queje. Por último se copia el bit más significativo al acarreo de salida. Dado que el tipo `unsigned` no es más que un vector de `std_logic`, al obtener un elemento del vector obtenemos un `std_logic` y por tanto no es necesario realizar ninguna conversión para asignarlo a `c_out`.



Realice el ejercicio 2

5.3. Restador de n bits

Aunque se podría realizar un circuito restador que implantase el algoritmo de la resta descrito en la sección 3.6, en la práctica se implanta la resta de $a-b$ como una suma de números con signo en la que se cambia al signo al sustraendo obteniendo su complemento a dos, es decir haciendo $a+(-b)$. Como recordará, la operación del complemento a dos es simplemente invertir todos los bits y sumar 1. Para invertir los bits se usa una serie de inversores. Para sumar uno, si tenemos en cuenta que el sumador en realidad hace la operación $a+b+c_0$, si hacemos que c_0 sea uno el sumador hará la operación $a+b+1$. Por tanto, si invertimos todos los bits de b y conectamos c_0 a uno, la operación realizada por el sumador será: $a+\overline{b}+1=a-b$. Teniendo todo esto en cuenta, un circuito para restar números de 4 bits quedaría tal como se muestra en la figura 5.9.

En la figura se ha usado un sumador con acarreo serie, pero se podría haber hecho la suma igualmente con un sumador con predicción de acarreo. En ambos casos tenga en cuenta que en la suma de números con signo en complemento a dos, el desbordamiento de la suma se detecta comprobando si los acarreos de los

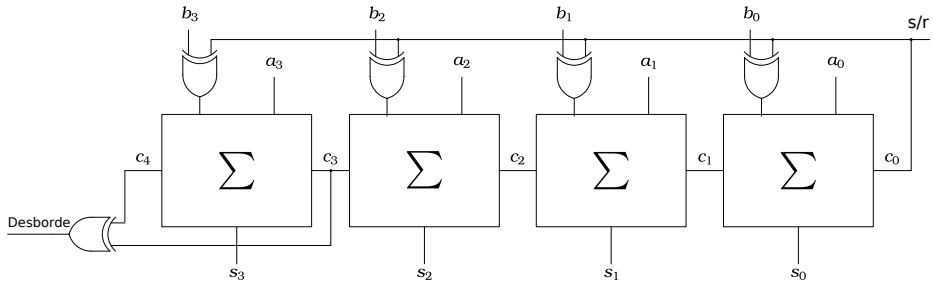


Figura 5.10: Sumador/Restador de 4 bits.

dos bits más significativos es distinto, tal como se expuso en la sección 3.9.2. Para detectar esta situación se usa una puerta XOR.

Realice los ejercicios 3 y 4

5.4. Sumador/Restador de n bits

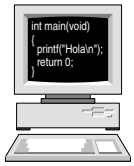
En un ordenador es necesario tanto hacer sumas como restas, pero como éstas no se hacen al mismo tiempo, podemos aprovechar un mismo circuito para que pueda hacer ambas operaciones en función de una señal de control. Para ello en primer lugar tenemos que conseguir que el número b pueda pasar al sumador invertido o sin invertir. Si recordamos la tabla de verdad de la puerta XOR:

i	b	s
0	0	0
0	1	1
1	0	1
1	1	0

podemos ver que cuando la entrada i vale cero, la salida s coincide con la entrada b, pero cuando i es igual a uno, la salida s es igual a la entrada b invertida. Teniendo esto en cuenta podemos usar una puerta XOR en lugar de un inversor para poder invertir o no la entrada b del sumador/restador en función de la operación que se desee hacer. Esto es lo que se ha hecho en la figura 5.10, donde como puede apreciar una entrada de las XOR se ha conectado a una señal de control a la que se ha denominado $\overline{s}/r$. Así, cuando esta señal sea cero la entrada b no se invertirá y cuando sea uno se invertirá.[2]

Para obtener el complemento a dos de b, además de invertir todos sus bits, hemos de sumar uno. Para ello, tal como hemos realizado en la sección anterior

[2]La notación usada para la señal $\overline{s}/r$ es muy común en los sistemas digitales. Cuando una señal de control tiene dos acciones asociadas en función de su valor, el nombre de la señal consta de dos símbolos separados por / y al símbolo que representa la función realizada cuando la señal vale cero se le pone una línea encima. En este caso el nombre $\overline{s}/r$ nos recuerda que el circuito realiza una suma cuando esta señal vale cero y una resta cuando vale 1.



basta con poner a uno la entrada c_0 del sumador; lo cual se consigue conectando la señal de control $\overline{s}/r$ a c_0, ya que ésta vale uno cuando queramos restar.

En definitiva, el circuito mostrado en la figura 5.10 realiza la operación $a+b+0$ cuando la entrada de control $\overline{s}/r = 0$ y la operación $a+\overline{b}+1 = a-b$ cuando $\overline{s}/r = 1$.

Realice los ejercicios 5 y 6

5.5. Multiplicadores

Existen varios circuitos capaces de realizar multiplicaciones. Los más rápidos son los combinacionales que vamos a estudiar en esta sección, aunque también necesitan mucha más lógica. En este tipo de multiplicadores todas las sumas y desplazamientos se hacen en paralelo. Para ahorrar *hardware* también se pueden realizar circuitos en los que las sumas parciales se realizan secuencialmente; pero dado que estos circuitos necesitan elementos de memoria dejaremos su estudio para más adelante.

El punto de partida es el algoritmo de multiplicación expuesto en la sección 3.6.3. Si queremos multiplicar dos números a y b de 4 bits, las operaciones a realizar son:

					a_3	a_2	a_1	a_0
				x	b_3	b_2	b_1	b_0
(pp_0)					$a_3 \cdot b_0$	$a_2 \cdot b_0$	$a_1 \cdot b_0$	$a_0 \cdot b_0$
(pp_1)				$a_3 \cdot b_1$	$a_2 \cdot b_1$	$a_1 \cdot b_1$	$a_0 \cdot b_1$	
(pp_2)			$a_3 \cdot b_2$	$a_2 \cdot b_2$	$a_1 \cdot b_2$	$a_0 \cdot b_2$		
(pp_3)	+	$a_3 \cdot b_3$	$a_2 \cdot b_3$	$a_1 \cdot b_3$	$a_0 \cdot b_3$			
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

Nótese que cada producto parcial pp_i se calcula como la AND entre el multiplicando y el bit i del multiplicador. Para obtener el producto final p es necesario sumar los cuatro productos parciales. Todas estas operaciones se pueden realizar con una matriz de sumadores y de puertas AND, tal como se muestra en la figura 5.11. En dicha figura, la primera fila de puertas calcula el producto parcial pp_0, la segunda el producto parcial pp_1 y así sucesivamente. El primer sumador calcula la suma entre los dos primeros productos parciales, es decir realiza $pp_0 + pp_1$. El resultado de esta operación se suma al producto parcial pp_2 en el segundo sumador y así sucesivamente. Fíjese que el acarreo de salida de cada sumador es necesario sumarlo al último bit del sumador siguiente.

5.5.1. Descripción en VHDL

La descripción en VHDL como es habitual se puede realizar a partir del circuito mostrado en la figura 5.11. Para simplificar la descripción usaremos el operador suma definido para el tipo `unsigned`. Para crear los productos parciales lo más fácil es crearlos todos con el mismo número de bits del resultado, añadiendo ceros a ambos lados para situar cada sumando en su posición. Así, la operación a realizar será:

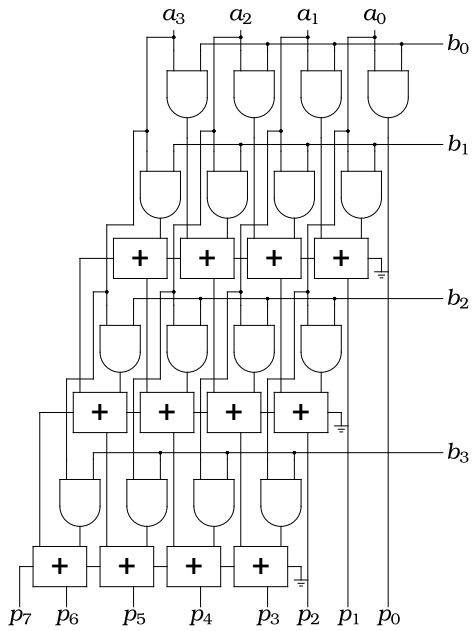


Figura 5.11: Multiplicador combinacional de 4 bits.

					a_3	a_2	a_1	a_0
				x	b_3	b_2	b_1	b_0
(pp_0)	0	0	0	0	$a_3 \cdot b_0$	$a_2 \cdot b_0$	$a_1 \cdot b_0$	$a_0 \cdot b_0$
(pp_1)	0	0	0	$a_3 \cdot b_1$	$a_2 \cdot b_1$	$a_1 \cdot b_1$	$a_0 \cdot b_1$	0
(pp_2)	0	0	$a_3 \cdot b_2$	$a_2 \cdot b_2$	$a_1 \cdot b_2$	$a_0 \cdot b_2$	0	0
(pp_3)	0	$a_3 \cdot b_3$	$a_2 \cdot b_3$	$a_1 \cdot b_3$	$a_0 \cdot b_3$	0	0	0
	p_7	p_6	p_5	p_4	p_3	p_2	p_1	p_0

En una primera descripción vamos a crear los productos parciales “a lo bruto”, para a continuación hacerlo de forma más elegante con bucles:

```
1  -- Multiplicador para números de 4 bits. Los productos
2  -- parciales se crean "a lo bruto".
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  entity Multipl4BitsBr is
9
10   port (
11     a, b : in  std_logic_vector(3 downto 0);   -- Entradas
```

```
12    p    : out std_logic_vector(7 downto 0));  -- Salida
13
14 end Multipl4BitsBr;
15
16 architecture behavioral of Multipl4BitsBr is
17   -- Productos parciales
18   signal pp0, pp1, pp2, pp3 : unsigned(7 downto 0);
19 begin  -- behavioral
20   pp0 <= "0000" & (a(3) and b(0)) & (a(2) and b(0)) &
21                   (a(1) and b(0)) & (a(0) and b(0));
22   pp1 <= "000"  & (a(3) and b(1)) & (a(2) and b(1)) &
23                   (a(1) and b(1)) & (a(0) and b(1)) & '0';
24   pp2 <= "00"   & (a(3) and b(2)) & (a(2) and b(2)) &
25                   (a(1) and b(2)) & (a(0) and b(2)) & "00";
26   pp3 <= '0'    & (a(3) and b(3)) & (a(2) and b(3)) &
27                   (a(1) and b(3)) & (a(0) and b(3)) & "000";
28
29   p <= std_logic_vector((pp0 + pp1) + (pp2 + pp3));
30
31 end behavioral;
```

Como puede observar se han creado cuatro señales de tipo `unsigned` para almacenar los productos parciales. Lo único a destacar es que cuando añadimos sólo un cero, al ser una señal del tipo `std_logic` hay que encerrarlo entre comillas simples. En cambio, cuando hemos añadido varios bits, al ser un vector, es necesario encerrarlo entre comillas dobles.

Por último, destacar cómo se ha realizado la suma:

```
p <= std_logic_vector((pp0 + pp1) + (pp2 + pp3));
```

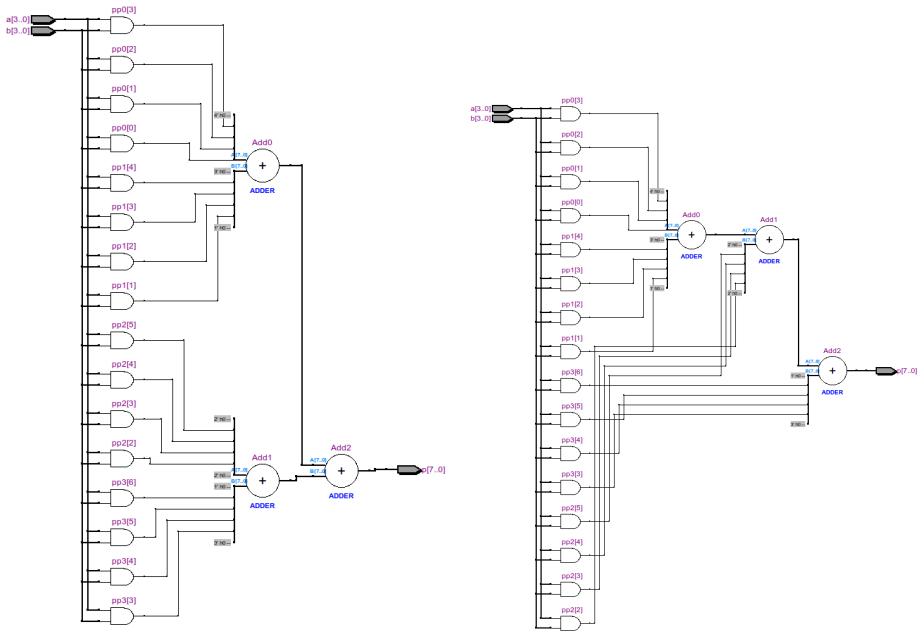
Nótese que se han agrupado con paréntesis los sumandos dos a dos. Esto permite guiar al sintetizador para que genere el circuito mostrado en la figura 5.12(a). Como sólo hay dos niveles de sumadores el retardo es de 14,956 ns. Si en cambio describimos la suma final como:

```
p <= std_logic_vector(pp0 + pp1 + pp2 + pp3);
```

El circuito sintetizado es el mostrado en la figura 5.12(b). Como ahora los sumadores se han generado en cascada, tenemos un retardo de 17,028 ns. Aunque la diferencia parezca pequeña, en multiplicadores de más bits será mucho más significativa. Aún así, poner un par de paréntesis no cuesta nada y así conseguimos un circuito mejor.

5.5.2. Descripción en VHDL usando bucles

Si el sumador es de más bits, el escribir las ecuaciones de los productos parciales puede ser un auténtico tostón. Sin embargo no es muy difícil describir el multipli-



(a) Con paréntesis (b) Sin paréntesis

Figura 5.12: Multiplicadores combinacionales de 4 bits sintetizados por Quartus II.

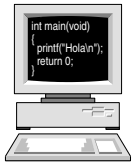
cador de la sección anterior usando bucles para crear dichos productos parciales; tal como se muestra a continuación:

```
1  -- Multiplicador para números de 4 bits
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5  use ieee.numeric_std.all;
6
7  entity Multipl4Bits is
8
9    port (
10     a, b : in  std_logic_vector(3 downto 0);   -- Entradas
11     p    : out std_logic_vector(7 downto 0));  -- Salida
12
13 end Multipl4Bits;
14
15 architecture behavioral of Multipl4Bits is
16   type t_vec_uns is array (0 to 3) of unsigned(7 downto 0);
```

```
17   signal pp : t_vec_uns;
18 begin  -- behavioral
19
20   ppgen: for i in 0 to 3 generate
21     b1: for j in 0 to i-1 generate
22       pp(i)(j) <= '0';
23     end generate b1;
24     b2: for j in i to i+3 generate
25       pp(i)(j) <= a(j-i) and b(i);
26     end generate b2;
27     b3: for j in i+3+1 to 7 generate
28       pp(i)(j) <= '0';
29     end generate b3;
30
31   end generate ppgen;
32
33   p <= std_logic_vector((pp(0) + pp(1)) + (pp(2) + pp(3)));
34
35 end behavioral;
```

Lo primero a destacar del código anterior es la creación de un vector de `unsigned` para almacenar los productos parciales. Para ello, tal como se mostró en la sección 4.5.2, en primer lugar es necesario definir un tipo y es lo que se ha realizado en la línea 16; en donde se ha creado un tipo para albergar vectores de 4 elementos de `unsigned` de 8 bits. En la línea siguiente se ha creado una variable de este tipo a la que se ha denominado `pp`.

Para generar los productos parciales se han usado varios bucles anidados. El primero genera el producto parcial `pp(i)` y dentro de este bucle tenemos otros tres. El primero, con etiqueta `b1`, genera los ceros en la parte menos significativa del producto parcial. El bucle `b2` genera los bits del producto parcial que no son cero ($a_0 \cdot b_0$, $a_1 \cdot b_0$, etc) y el último bucle (`b3` genera los ceros en la parte más significativa del producto parcial.



Realice el ejercicio 7

Por último, la suma final de los productos parciales se ha realizado igual que antes.

5.5.3. *Descripción en VHDL en alto nivel*

Al igual que con la suma, los tipos `signed` y `unsigned` soportan la operación de multiplicación, con lo que la descripción se simplifica en gran manera:

```
1 -- Multiplicador para números de 4 bits
2
3 library ieee;
4 use ieee.std_logic_1164.all;
5 use ieee.numeric_std.all;
```

```
6
7  entity Multipl4BitsAN is
8
9    port (
10     a, b : in  std_logic_vector(3 downto 0);   -- Entradas
11     p    : out std_logic_vector(7 downto 0));  -- Salida
12
13 end Multipl4BitsAN;
14
15 architecture behavioral of Multipl4BitsAN is
16 begin  -- behavioral
17
18   p <= std_logic_vector(unsigned(a)*unsigned(b));
19
20 end behavioral;
```

5.6. Sumador de números en BCD natural

Tal como se expuso en la sección 3.10.1, es posible también sumar números en BCD siguiendo un sencillo algoritmo consistente en sumar dígito a dígito y si el resultado es inválido o hay un acarreo al siguiente bit sumar 6 al resultado anterior. Para ello en primer lugar es necesario diseñar un circuito que detecte si un número BCD es inválido. Si partimos de la tabla de verdad mostrada en la figura 5.13 podemos obtener las ecuaciones lógicas a partir del diagrama de Karnaugh mostrado también en la figura:

$$I = d_3 \cdot d_2 + d_3 \cdot d_1$$

Según el algoritmo de la suma en BCD hay que sumar 6 cuando o bien el resultado sea un dígito BCD inválido o bien cuando se produzca un acarreo a la cifra siguiente. Además, cuando sumamos 6 también se produce un acarreo a la cifra siguiente, por lo que la señal que nos indica que hay que sumar 6 también nos indica que hay que enviar un acarreo a la cifra siguiente. Por ello a esta señal la denominamos c_{out} y su ecuación lógica es:

$$c_{out} = c_{outBin} + I = c_{outBin} + d_3 \cdot d_2 + d_3 \cdot d_1$$

En donde la señal c_{outBin} es la que nos indica si se ha producido un acarreo en la suma de las dos cifras BCD.

Teniendo todo esto en cuenta, la estructura del sumador BCD para un dígito es la mostrada en la figura 5.14. En ella se ha usado un primer sumador binario para realizar la suma de los dos dígitos BCD a y b. La salida de este sumador se conecta a otro en el que una de sus entradas está controlada por la señal c_{out}, de forma que si esta señal vale cero se suma 0 y si vale 1 se suma 6. Con esto, la salida s del segundo sumador contendrá la salida BCD corregida.

(a) Tabla de verdad

d_3	d_2	d_1	d_0	I
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

(b) Diagrama de Karnaugh

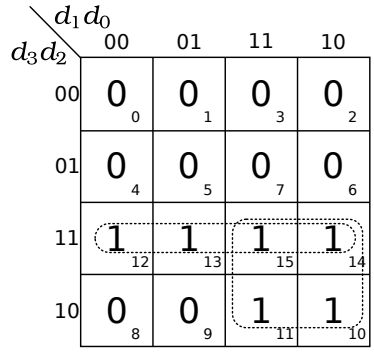


Figura 5.13: Detección de código BCD inválido.

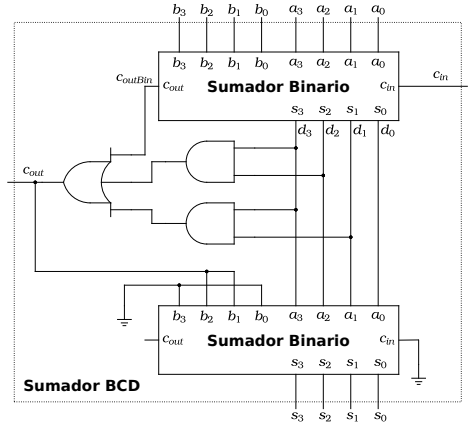


Figura 5.14: Sumador BCD de un dígito.

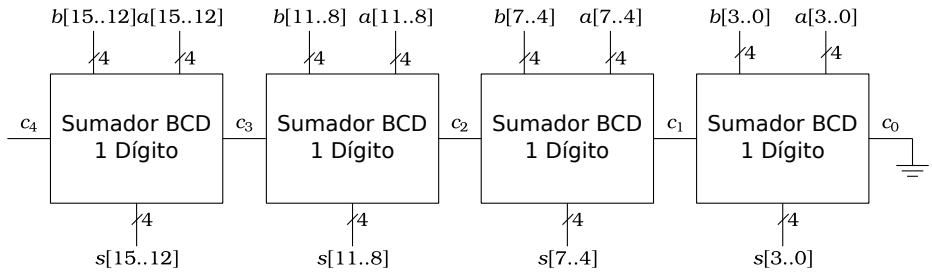


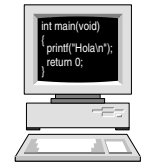
Figura 5.15: Sumador BCD de cuatro dígitos.

El circuito de la figura 5.14 se puede conectar en cascada para realizar sumadores BCD de varios dígitos, de la misma forma que se hizo con los sumadores binarios. En este caso es preciso conectar los acarreos en serie entre los distintos sumadores. El circuito completo para sumar números BCD de cuatro cifras se muestra en la figura 5.15

Si observa la figura 5.14 se dará cuenta de los dos inconvenientes de este circuito frente al sumador binario:

- El circuito necesita más puertas lógicas. Por cada dígito BCD se necesitan dos sumadores binarios y tres puertas lógicas. Si además tenemos en cuenta que para representar un número BCD se necesitan más bits que en binario, el resultado es que hace falta mucho más del doble de puertas para implantar un sumador BCD que un sumador binario.
- Como el cálculo de cada dígito del resultado necesita dos sumas y el acarreo necesita dos niveles más de puertas por cada dígito, el circuito es también más lento que el sumador binario.

Estos inconvenientes hacen que en la práctica se evite trabajar con números en BCD si es necesario hacer una gran cantidad de cálculos con ellos. En estos casos es preferible trabajar en binario y hacer una conversión de binario a BCD y viceversa cuando sea necesario. Más adelante estudiaremos un circuito para realizar dicha conversión.



Realice el ejercicio 8

5.7. Ejercicios

1. Describa en VHDL un sumador con predicción de acarreo genérico de n bits. Para ello implante la unidad de predicción de acarreos a partir de las ecuaciones (5.3–5.6) mediante una sentencia **`for ... generate`**. Use 4 bits como valor por defecto para el parámetro.
2. Describa en VHDL un sumador genérico de N bits usando el operador + con tipos `unsigned`. Use 4 bits como valor por defecto para el parámetro.
3. Describa en VHDL un restador de 4 bits. Use un sumador con acarreo serie para realizar la suma.

4. Modifique el resultado del ejercicio anterior para describir un restador de n bits. Use 4 como valor por defecto para el parámetro genérico.
5. Modifique el resultado del ejercicio 3 para describir un sumador/restador de 4 bits.
6. Modifique el resultado del ejercicio 4 para describir un sumador/restador de n bits. Use 4 como valor por defecto para el parámetro genérico.
7. Modifique el multiplicador mostrado en la sección 5.5.2 para multiplicar dos números de 8 bits.
8. Describa en VHDL un sumador en BCD para números de dos dígitos. Para ello se creará en primer lugar un bloque denominado `SumadorBCD1Digito` para sumar un dígito BCD. Para realizar este bloque puede usar el sumador binario de 4 bits descrito en este capítulo. Una vez diseñado este bloque se realizará una descripción estructural en la que se instanciarán dos bloques `SumadorBCD1Digito` para realizar el sumador de dos dígitos.

CAPÍTULO 6

Bloques Combinacionales

Además de los circuitos aritméticos estudiados en el capítulo anterior, existen una serie de bloques combinacionales básicos que son muy utilizados en la práctica. En este capítulo se van a estudiar los más usados en la práctica.

6.1. Multiplexor

Un multiplexor permite elegir cuál de las varias entradas de datos de que dispone se copia a su salida, todo ello en función de una serie de entradas de control.

Un multiplexor dispone de n entradas de control, 2^n entradas de datos y una única salida. A modo de ejemplo, en la figura 6.1 se muestra un multiplexor de 4 a 1, es decir, un multiplexor con 4 entradas de datos denominadas E3, E2, E1 y E0 y una salida S. Al disponer de 4 entradas de datos, son necesarias $\log_2 n = \log_2 4 = 2$ entradas de control, denominadas C1 y C0.

El funcionamiento del multiplexor es muy simple: el número binario formado por las entradas de control (C1 C0 en el ejemplo de la figura 6.1) indica el número de la entrada que se conecta a la salida. Así, si en el circuito de la figura 6.1 C1=1 y C0=0, como el número binario formado es un dos, la salida tomará el mismo valor que la entrada E2.

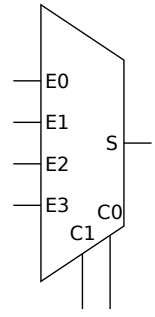


Figura 6.1: Multiplexor de 4 a 1.

C1	C0	E3	E2	E1	E0	S
0	0	x	x	x	0	0
0	0	x	x	x	1	1
0	1	x	x	0	x	0
0	1	x	x	1	x	1
1	0	x	0	x	x	0
1	0	x	1	x	x	1
1	1	0	x	x	x	0
1	1	1	x	x	x	1

(a) Tabla de verdad completa.

C1	C0	S
0	0	E0
0	1	E1
1	0	E2
1	1	E3

(b) Tabla de verdad simplificada

Cuadro 6.1: Tablas de verdad del multiplexor de 4 a 1

La tabla de verdad del circuito se muestra en el cuadro 6.1(a). En ella las x en las entradas indican que dicha entrada no tiene ninguna influencia sobre la salida. Así, la primera línea nos dice que cuando C1, C0 y E0 valen cero, la salida S vale cero; independientemente del valor del resto de entradas. Este comportamiento nos permite escribir la tabla de una forma mucho más simple, tal como se muestra en el cuadro 6.1(b). En ella la primera línea nos dice que cuando C1 y C0 valen 0 la salida toma el valor de la entrada E0. En estos casos conviene no dejarse engañar, pues aunque en la tabla sólo están tabuladas las entradas de control C1 y C0, el circuito sigue teniendo seis entradas.

Cuando tenemos una tabla de verdad como la mostrada en el cuadro 6.1(b), en el que la salida depende de una serie de entradas, la forma de obtener la expresión lógica es multiplicar el minitérmino asociado a la fila por la entrada que aparece en la columna de la salida para dicha fila. Así, la expresión lógica del multiplexor de 4 a 1 es:

$$S = \overline{C1} \cdot \overline{C0} \cdot E0 + \overline{C1} \cdot C0 \cdot E1 + C1 \cdot \overline{C0} \cdot E2 + C1 \cdot C0 \cdot E3$$

6.1.1. Multiplexores para entradas de n bits

Es muy frecuente tener que multiplexar en lugar de varias señales de un solo bit, varias señales de n bits. En este caso basta con colocar varios multiplexores en paralelo controlados por las mismas señales de control, tal como se muestra en la figura 6.2.

6.1.2. Descripción en VHDL

Aunque un multiplexor podría describirse a partir de sus ecuaciones lógicas, el lenguaje VHDL dispone de la construcción **`with ... select`** para ello. Por ejemplo, el multiplexor de 4 a 1 estudiado hasta ahora se describe como:

```
1 -- Multiplexor de 4 a 1
2
```

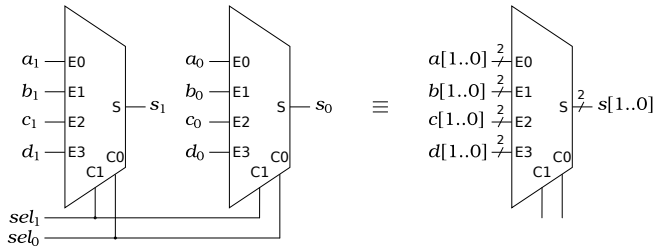
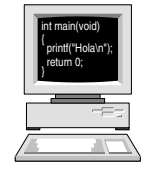


Figura 6.2: Multiplexor de 4 a 1 de dos bits.

```
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity Mux4a1 is
7
8    port (
9      e3, e2, e1, e0 : in  std_logic; -- Entradas de datos
10                                     -- Entradas de control
11     sel            : in  std_logic_vector(1 downto 0);
12     s              : out std_logic);           -- Salida
13
14 end Mux4a1;
15
16 architecture behavioral of Mux4a1 is
17
18 begin  -- behavioral
19
20   with sel select
21     s <=
22     e0  when "00",
23     e1  when "01",
24     e2  when "10",
25     e3  when "11",
26     '0' when others;
27
28 end behavioral;
```

Si queremos describir un multiplexor de más entradas basta con tomar como modelo la descripción anterior, teniendo en cuenta la relación entre las n entradas de control y las 2^n entradas de datos. Por otro lado, si se desea describir un multiplexor para entradas de varios bits, basta con cambiar las señales de datos y la salida de `std_logic` a `std_logic_vector`.



Realice los ejercicios 1, 2 y 3

(a) Tabla de verdad

C1	C0	S3	S2	S1	S0
0	0	0	0	0	E
0	1	0	0	E	0
1	0	0	E	0	0
1	1	E	0	0	0

(b) Símbolo

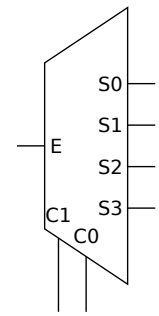


Figura 6.3: Demultiplexor de 1 a 4.

6.2. Demultiplexores

Como su propio nombre indica, el demultiplexor es un circuito con la funcionalidad contraria al multiplexor. El circuito dispone de una entrada de datos, n de control y 2^n salidas, de forma que el valor de la entrada se copia a la salida cuyo número indican las entradas de control. El resto de salidas permanecen a cero.

En la figura 6.3 se muestra la tabla de verdad de un demultiplexor de 1 a 4 y su símbolo. Como se puede apreciar, cuando la señal de control toma el valor binario 00, la salida $S0$ toma el valor de la entrada E; cuando la señal de control es 01, es la salida $S1$ la que toma el valor de E y así sucesivamente.

A partir de la tabla de verdad es fácil obtener las ecuaciones lógicas de las cuatro salidas. Como cada salida vale E sólo para una fila de la tabla, el valor de la salida será el producto entre la entrada E y el minitérmino correspondiente a la fila en la que aparece dicha entrada:

$$\begin{aligned} S0 &= \overline{C1} \cdot \overline{C0} \cdot E \\ S1 &= \overline{C1} \cdot C0 \cdot E \\ S2 &= C1 \cdot \overline{C0} \cdot E \\ S3 &= C1 \cdot C0 \cdot E \end{aligned}$$

6.2.1. Demultiplexores para señales de n bits

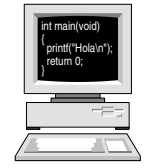
Al igual que con el multiplexor, es posible demultiplexar señales de varios bits mediante el uso de varios demultiplexores en paralelo y controlarlos por las mismas señales de selección.

6.2.2. *Descripción en VHDL*

Al igual que con el multiplexor, aunque el demultiplexor podría describirse mediante las ecuaciones lógicas que acabamos de obtener, existen construcciones en el lenguaje que facilitan la descripción. En este caso la más apropiada es la asignación condicional, tal como se muestra en el código siguiente:

```
1  -- Demultipexor de 1 a 4 de 1 bit
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity DeMux1a4 is
7
8    port (
9      e   : in  std_logic;                -- Entrada de datos
10                                        -- Entrada de control
11     sel : in  std_logic_vector(1 downto 0);
12     s0, s1, s2, s3  : out std_logic);           -- Salidas
13
14 end DeMux1a4;
15
16 architecture behavioral of DeMux1a4 is
17
18 begin  -- behavioral
19
20   s0 <= e when sel = "00" else
21         '0';
22   s1 <= e when sel = "01" else
23         '0';
24   s2 <= e when sel = "10" else
25         '0';
26   s3 <= e when sel = "11" else
27         '0';
28
29 end behavioral;
```

Como puede apreciar, cada salida se asigna a la entrada `e` solo cuando la entrada de selección `sel` toma el valor correspondiente a la salida. Así, a la salida `s0` se le asigna la entrada `e` sólo cuando la entrada de selección `sel` es igual a `"00"`.



Realice el ejercicio 4

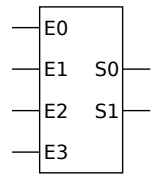
6.3. Codificadores

Un codificador es un circuito que a su salida nos da un número binario que indica cuál de sus entradas está activa. Si el codificador dispone de n salidas, lo

(a) Tabla de verdad

E3	E2	E1	E0	S1	S0
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1
Resto combinaciones				X	X

(b) Símbolo



(c) Diagramas de Karnaugh

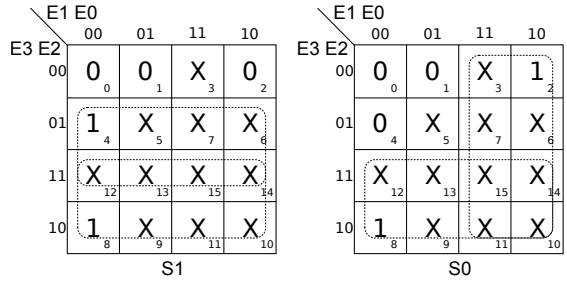


Figura 6.4: Codificador de 4 entradas.

lógico es que disponga de $m = 2^n$ entradas, ya que éstos son los números binarios que se pueden formar con n bits. Se dice entonces que el codificador es **completo**. No obstante, también se pueden crear codificadores en los que se cumple que el número de entradas m es menor que 2^n. En estos casos se dice que el codificador es **incompleto**.

Por ejemplo, en la figura 6.4 se muestran la tabla de verdad y el símbolo de un codificador completo de 4 entradas. En este codificador se ha supuesto que sólo puede estar activa una entrada a la vez, por lo que para simplificar las salidas se han puesto a *don't care* en todas las combinaciones de entradas en las que hay varias de ellas activas, tal como puede apreciarse en la tabla de verdad. A este tipo de codificadores se les denomina **codificadores sin prioridad**.

Las ecuaciones lógicas del circuito pueden obtenerse a partir de los diagramas de Karnaugh. Nótese que al existir un montón de filas en la tabla como *don't care* los diagramas están llenos de X, lo que nos permite crear grupos muy grandes que dan lugar a ecuaciones muy simples, las cuales son:

$$S1 = E3 + E2$$
$$S0 = E3 + E1$$

Nótese que la entrada $E0$ no aparece en las ecuaciones. Esto es debido a que no influye en la tabla, pues como puede apreciar en las dos primeras filas, independientemente del valor de $E0$ la salida vale 0. Como en el resto de las filas de la tabla

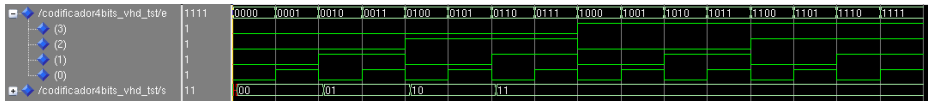


Figura 6.5: Diagrama de tiempos de un codificador sin prioridad de cuatro entradas.

las salidas están a X, el resultado es que dicha entrada no influye en el valor de las salidas. La consecuencia práctica de todo esto es que es imposible saber cuándo la entrada *E*0 está activa. Para ello se añade a los codificadores una salida denominada *I* que se pone a 1 cuando todas las entradas están inactivas. La tabla de verdad en este caso es:

E3	E2	E1	E0	S1	S0	I
0	0	0	0	0	0	1
0	0	0	1	0	0	0
0	0	1	0	0	1	0
0	1	0	0	1	0	0
1	0	0	0	1	1	0
Resto combinaciones				X	X	0

La ecuación de la salida *I*, al sólo valer 1 en la primera fila, coincide con el minitérmino asociado a ésta, es decir:

$$I = \overline{E3} \cdot \overline{E2} \cdot \overline{E1} \cdot \overline{E0} \tag{6.1}$$

Si observamos el funcionamiento del circuito mostrado en la figura 6.5 podemos apreciar que cuando se activan varias entradas a la vez el circuito hace cosas raras. Por ejemplo cuando se activan E2 y E1 ((0110 en la figura) la salida del circuito se pone a 11_2, lo cual quiere decir que se ha activado la entrada E3; nada más lejos de la realidad. Lo que ocurre en este caso es que hemos dejado estas combinaciones de entradas como *don't care* pensando que no se iban a dar en la práctica. El problema es que si se dan el circuito hace lo que le da la gana, que es precisamente lo que habíamos especificado en la tabla de verdad al poner las X.[1]

Descripción en VHDL

La descripción en VHDL de un decodificador puede realizarse fácilmente a partir de la tabla de verdad mediante la sentencia **`with ... select`**, tal como se muestra en el listado siguiente:

```
1 -- Codificador de 4 entradas
```

[1] Cuando se pone una salida a X (*don't care*) en una fila de la tabla de verdad lo que queremos decir realmente es que nos da igual lo que valga la salida para esa combinación de entradas pues dicha combinación no se va a producir nunca. El problema lo tendremos cuando se produzca esta combinación de entradas, pues la salida valdrá cualquier cosa.

```
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity Codificador4Entradas is
7
8    port (
9      e : in  std_logic_vector(3 downto 0);  -- Entradas
10     s : out std_logic_vector(1 downto 0);  -- Salidas
11     i : out std_logic);
12
13 end Codificador4Entradas;
14
15 architecture behavioral of Codificador4Entradas is
16
17 begin  -- behavioral
18
19   with e select
20     s <=
21     "00" when "0000",
22     "00" when "0001",
23     "01" when "0010",
24     "10" when "0100",
25     "11" when "1000",
26     "--" when others;
27
28   i <= '1' when e = "0000" else
29        '0';
30
31 end behavioral;
```

Lo único destacable de esta descripción se encuentra en la línea 26, en la que se asigna a la salida el valor *don't care* para el resto de combinaciones de la entrada e que no se han especificado en las cinco líneas anteriores. Nótese que en VHDL el valor *don't care* se representa mediante el símbolo '-'.

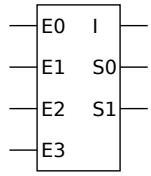
6.3.1. Codificador con prioridad

Si queremos que un codificador funcione correctamente cuando se activen varias entradas a la vez tenemos en primer lugar que pensar qué hacer en ese caso, pues recordemos que un codificador nos codifica en su salida el valor de la entrada activa. Una posibilidad es codificar en la salida el número de la entrada activa mayor. Así si se activan por ejemplo las entradas E1 y E3, la salida se pondrá a 3. En estos casos decimos que el codificador es **con prioridad a la entrada mayor**.

(a) Tabla de verdad

E3	E2	E1	E0	S1	S0	I
0	0	0	0	0	0	1
0	0	0	1	0	0	0
0	0	1	X	0	1	0
0	1	X	X	1	0	0
1	X	X	X	1	1	0

(b) Símbolo



(c) Diagramas de Karnaugh

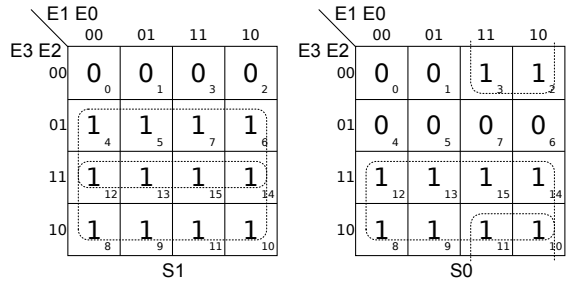


Figura 6.6: Codificador de 4 entradas con prioridad.

En la figura 6.6 se muestra la tabla de verdad de un codificador de 4 entradas con prioridad a la entrada mayor. En este caso las X en la columna de las entradas indican que para cualquier valor de esa entrada las salidas valen lo mismo. En realidad esto no es más que una manera de escribir la tabla de verdad de una manera más compacta, pues por ejemplo la segunda fila en realidad es equivalente a las dos filas:

E3	E2	E1	E0	S1	S0	I
0	0	1	0	0	1	0
0	0	1	1	0	1	0

A partir de los diagramas de Karnaugh mostrados en la figura 6.6 es fácil obtener las ecuaciones del codificador con prioridad:

$$\begin{aligned} S1 &= E3 + E2 \\ S0 &= E3 + \overline{E2} \cdot E1 \\ I &= \overline{E3} \cdot \overline{E2} \cdot \overline{E1} \cdot \overline{E0} \end{aligned}$$

Como puede observar, las ecuaciones son un poco más complejas al considerarse la prioridad, aunque tampoco tanto.

En este caso el funcionamiento del circuito es el mostrado en la figura 6.7, donde se puede apreciar que ahora cuando se activan varias entradas a la vez la salida indica el número de la entrada activa mayor.

Figura 6.7: Diagrama de tiempos de un codificador con prioridad de cuatro entradas.

6.3.2. descripción en VHDL

La descripción en VHDL no puede incluir las X de las entradas, por lo que tenemos dos posibilidades: o especificar la tabla de verdad completa con la sentencia **with ... select** o realizar una asignación condicional, que es lo más cómodo. En este caso la descripción queda como sigue:

```
1  -- Codificador de 4 entradas con prioridad a la mayor
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity Codificador4EntradasPrioMayor is
7
8    port (
9      e : in  std_logic_vector(3 downto 0);  -- Entradas
10     s : out std_logic_vector(1 downto 0);  -- Salidas
11     i : out std_logic);
12
13 end Codificador4EntradasPrioMayor;
14
15 architecture behavioral of Codificador4EntradasPrioMayor is
16
17 begin  -- behavioral
18
19   s <= "11" when e(3) = '1' else
20        "10" when e(2) = '1' else
21        "01" when e(1) = '1' else
22        "00" when e(0) = '1' else
23        "00";
24
25   i <= '1' when e = "0000" else
26        '0';
27
28 end behavioral;
```

Como la asignación condicional mediante **when ... else** asigna el valor asociado a la primera condición que se cumpla, la prioridad se gestiona adecuada-

(a) Tabla de verdad

E1	E0	S3	S2	S1	S0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

(b) Símbolo

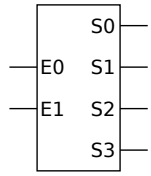


Figura 6.8: Decodificador de 2 entradas y 4 salidas.

mente. Así cuando se activen a la vez `e(2)` y `e(1)`, como primero se comprueba si `e(2) = '1'`, entonces se asignará a `s` el valor `"10"`; no evaluándose el resto de condiciones de la sentencia.

Realice los ejercicios 5 y 6.

6.4. Decodificadores

Su funcionamiento, como habrá podido adivinar por el nombre, es el contrario al codificador. El circuito dispone de n entradas y 2^n salidas de forma que pone a uno la salida cuyo número aparece en las n entradas. La tabla de verdad y el símbolo del circuito para un decodificador de 2 entradas y 4 salidas se ilustran en la figura 6.8.

Las ecuaciones lógicas son fáciles de obtener a partir de la tabla de verdad:

$$\begin{aligned} S0 &= \overline{E1} \cdot \overline{E0} \\ S1 &= \overline{E1} \cdot E0 \\ S2 &= E1 \cdot \overline{E0} \\ S3 &= E1 \cdot E0 \end{aligned}$$

La descripción en VHDL de este circuito puede realizarse o bien a partir de las ecuaciones lógicas anteriores o bien implantando la tabla de verdad mediante la sentencia **`with ... select`**. Ambas se dejan como ejercicio.

Realice el ejercicio 7

6.5. Comparadores

Un comparador nos permite comparar dos números de n bits para averiguar si son iguales o si uno es mayor que el otro. Aunque es un circuito puramente combinacional, cuando el número de bits de los números de entrada es elevado la tabla de verdad se hace inmanejable.

Otra alternativa para implantar este tipo de circuitos es realizarlo de forma iterativa, tal como se hace con los sumadores. Para ello en lo primero que hay que pararse a pensar es en cómo realizamos nosotros la comparación de dos números

(a) Tabla de verdad

a	b	a_ma_b	a_ig_b	a_me_b
0	0	a_ma_b_i	a_ig_b_i	a_me_b_i
0	1	0	0	1
1	0	1	0	0
1	1	a_ma_b_i	a_ig_b_i	a_me_b_i

(b) Símbolo

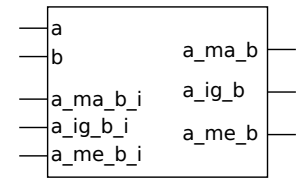


Figura 6.9: Comparador de 1 bit.

de n cifras. Por ejemplo si nos piden que comparemos el número 270 con el 243, empezamos comparando los dos dígitos más significativos. Como ambos son iguales, no podemos sacar aún una conclusión, por lo que pasamos a estudiar el siguiente dígito. En este caso, como el 7 es mayor que el 4 ya podemos decir que el 270 es mayor que el 243, sin necesidad de seguir mirando el resto de los dígitos. Si comparamos el 273 con el 273, llegaremos al dígito menos significativo sin encontrar uno mayor que otro, con lo que concluimos que ambos números son iguales.

El proceso anterior se puede realizar también para comparar números binarios. Así, si nos piden comparar el 1001 con el 1010, empezaremos por el bit más significativo y nos daremos cuenta que en ambos números es un uno, por lo que pasaremos a estudiar el bit 2. Con este bit nos pasará lo mismo, por lo que pasaremos a estudiar el bit 1 y ahí nos daremos cuenta que el segundo número es mayor que el primero pues dicho bit 1 vale uno para el 1010 y cero para el 1001.

6.5.1. Comparador de 1 bit

Para implantar un comparador siguiendo esta técnica lo primero que necesitamos es un comparador de 1 bit. La tabla de verdad y el símbolo del comparador se ilustra en la figura 6.9. Como puede observar, el circuito dispone de tres salidas para indicarnos el resultado de la comparación: `a_ma_b` se pondrá a 1 cuando la entrada `a` sea mayor que `b`, `a_me_b` se pondrá a 1 cuando la entrada `a` sea menor que `b` y `a_ig_b` se pondrá a 1 cuando ambas entradas sean iguales. Por otro lado, el circuito dispone de cinco entradas: `a` y `b` que son los dos bits que se comparan y `a_ma_b_i`, `a_ig_b_i` y `a_me_b_i` por las que se introduce el resultado de la comparación de los bits menos significativos. Estas entradas, como puede apreciar en la tabla de verdad sólo se usan cuando las entradas `a` y `b` son iguales, pues en este caso el circuito no puede saber el resultado de la comparación y tiene que "preguntárselo" a los comparadores de los bits de menor peso, tal como veremos a continuación.

Las ecuaciones lógicas del comparador se extraen fácilmente de su tabla de

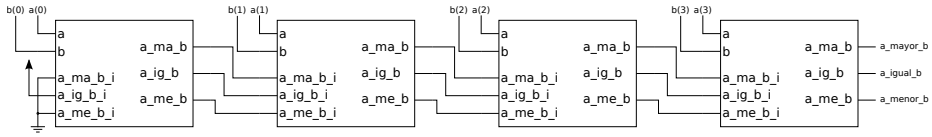


Figura 6.10: Comparador de 4 bits.

verdad:

$$\begin{aligned} a\_ma\_b &= \overline{a}\cdot\overline{b}\cdot a\_ma\_b\_i + a\cdot\overline{b} + a\cdot b\cdot a\_ma\_b\_i = a\cdot\overline{b} + \overline{a\oplus b}\cdot a\_ma\_b\_i \\ a\_ig\_b &= \overline{a}\cdot\overline{b}\cdot a\_ig\_b\_i + a\cdot b\cdot a\_ig\_b\_i = \overline{a\oplus b}\cdot a\_ig\_b\_i \\ a\_me\_b &= \overline{a}\cdot\overline{b}\cdot a\_me\_b\_i + \overline{a}\cdot b + a\cdot b\cdot a\_me\_b\_i = \overline{a}\cdot b + \overline{a\oplus b}\cdot a\_me\_b\_i \end{aligned}$$

6.5.2. *Comparador de n bits*

Para implantar un comparador de n bits conectamos en cascada n comparadores de un bit, de forma que las salidas del comparador del bit de menor peso se conectan a las entradas del de mayor peso, tal como se muestra en la figura 6.10, en la que se muestra un comparador de cuatro bits.

El funcionamiento del circuito se basa en la forma en que nosotros comparamos los números. El comparador que hay más a la derecha compara los dos bits más significativos, de forma que si son distintos no depende de nadie para obtener el resultado, activando entonces la salida `a_mayor_b` o `a_menor_b` según los valores de `a(3)` y `b(3)`. Si por el contrario `a(3)` y `b(3)` son iguales, este comparador no puede saber el resultado por si mismo, teniendo entonces que "preguntárselo" al comparador del bit 2. En este caso, tal como podemos apreciar en la tabla de verdad de la figura 6.9, el comparador del bit 3 se limita a copiar en sus salidas el valor de las entradas `a_ma_b_i`, `a_ig_b_i` y `a_me_b_i`. Si todos los bits son iguales, el comparador del bit 0 copia sus entradas `a_ma_b_i`, `a_ig_b_i` y `a_me_b_i` a sus salidas. Como en este caso los números `a` y `b` son iguales, es necesario conectar estas entradas tal como se muestra en la figura, de forma que la salida del comparador sea `a_ma_b_i=0`, `a_ig_b_i=1` y `a_me_b_i=0`.

6.5.3. *Descripción en VHDL*

La descripción en VHDL se realiza creando en primer lugar el comparador de 1 bit. Para ello basta con describir el circuito a partir de sus ecuaciones lógicas:

```
1 -- Comparador de 1 bit
2
3 library ieee;
4 use ieee.std_logic_1164.all;
5
6 entity Comparador1Bit is
```

```
7
8   port (
9     a, b     : in  std_logic;   -- Entradas
10    a_ma_b_i : in  std_logic;   -- Entradas del comparador
11    a_ig_b_i : in  std_logic;   -- anterior
12    a_me_b_i : in  std_logic;
13    a_ma_b   : out std_logic;   -- Salidas
14    a_ig_b   : out std_logic;
15    a_me_b   : out std_logic);
16
17 end Comparador1Bit;
18
19 architecture behavioral of Comparador1Bit is
20
21 begin  -- behavioral
22
23   a_ma_b <= (a and not b) or (not(a xor b) and a_ma_b_i);
24   a_ig_b <= not(a xor b) and a_ig_b_i;
25   a_me_b <= (not a and b) or (not(a xor b) and a_me_b_i);
26
27 end behavioral;
```

Para crear un comparador de n bits se instancian n comparadores de 1 bit. Para ello se usa una descripción estructural en la que mediante un **for ... generate** se instancian los n comparadores, de la misma forma que se hizo para el sumador. Así, si queremos crear un comparador de 4 bits como el de la figura, basta con hacer:

```
1  -- Comparador de 4 bits a partir de comparadores de 1 bit
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity Comparador4Bits is
7
8    port (
9      a, b   : in  std_logic_vector(3 downto 0);  -- Entradas
10     a_ma_b : out std_logic;                     -- a mayor que b
11     a_ig_b : out std_logic;                     -- a igual a b
12     a_me_b : out std_logic);                    -- a menor que b
13
14 end Comparador4Bits;
15
16 architecture structural of Comparador4Bits is
17   -- Señales para conectar los comparadores de 1 bit en cascada
```

```
18    signal a_ma_b_int : std_logic_vector(4 downto 0);
19    signal a_ig_b_int : std_logic_vector(4 downto 0);
20    signal a_me_b_int : std_logic_vector(4 downto 0);
21
22    component Comparador1Bit
23      port (
24        a, b     : in  std_logic;
25        a_ma_b_i : in  std_logic;
26        a_ig_b_i : in  std_logic;
27        a_me_b_i : in  std_logic;
28        a_ma_b   : out std_logic;
29        a_ig_b   : out std_logic;
30        a_me_b   : out std_logic);
31    end component;
32
33  begin  -- structural
34
35    -- Las entradas del comparador anterior del comparador
36    -- para el bit 0 han de conectarse así:
37    a_ma_b_int(0) <= '0';
38    a_ig_b_int(0) <= '1';
39    a_me_b_int(0) <= '0';
40
41    Comps : for i in 0 to 3 generate
42      -- instance "Comparador1Bit_1"
43      Comparador1Bit_1 : Comparador1Bit
44        port map (
45          a        => a(i),
46          b        => b(i),
47          a_ma_b_i => a_ma_b_int(i),
48          a_ig_b_i => a_ig_b_int(i),
49          a_me_b_i => a_me_b_int(i),
50          a_ma_b   => a_ma_b_int(i+1),
51          a_ig_b   => a_ig_b_int(i+1),
52          a_me_b   => a_me_b_int(i+1));
53    end generate Comps;
54
55    -- Las salidas del último comparador son las salidas
56    -- del bloque
57    a_ma_b <= a_ma_b_int(4);
58    a_ig_b <= a_ig_b_int(4);
59    a_me_b <= a_me_b_int(4);
60
61  end structural;
```

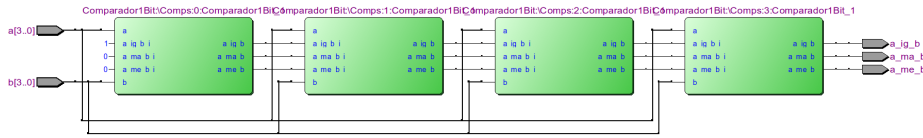
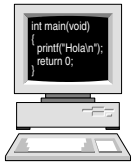


Figura 6.11: Comparador de 4 bits.



Realice el ejercicio 8

El circuito generado por el sintetizador de Quartus se muestra en la figura 6.11.

Descripción en alto nivel

Usando el paquete `ieee.std_numeric` es posible comparar números con signo y sin signo, tal como se muestra a continuación:

```
1  -- Comparador de 4 bits descrito en alto nivel
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5  use ieee.numeric_std.all;
6
7  entity Comparador4BitsAN is
8
9    port (
10     a, b   : in  std_logic_vector(3 downto 0);  -- Entradas
11     a_ma_b : out std_logic;                     -- a mayor que b
12     a_ig_b : out std_logic;                     -- a igual a b
13     a_me_b : out std_logic);                    -- a menor que b
14
15 end Comparador4BitsAN;
16
17 architecture behavioral of Comparador4BitsAN is
18
19 begin  -- behavioral
20
21   a_ma_b <= '1' when unsigned(a) > unsigned(b) else
22             '0';
23   a_ig_b <= '1' when unsigned(a) = unsigned(b) else
24             '0';
25   a_me_b <= '1' when unsigned(a) < unsigned(b) else
26             '0';
27 end behavioral;
```

6.6. Ejercicios

1. Describa en VHDL un multiplexor de 8 a 1 de 1 bit.
2. Describa en VHDL un multiplexor de 8 a 1 de 8 bits.
3. Describa en VHDL un multiplexor de 8 a 1 de N bits, siendo N un parámetro genérico.
4. Repita los ejercicios anteriores pero para describir demultiplexores en lugar de multiplexores.
5. Describa en VHDL un codificador de 4 entradas con prioridad a la entrada menor.
6. Describa en VHDL un codificador de 8 entradas sin prioridad. Repita el ejercicio para describir el mismo codificador con prioridad a la entrada mayor.
7. Describa en VHDL un decodificador de 2 entradas y 4 salidas. La descripción ha de realizarse de dos modos: implantando las ecuaciones lógicas e implantando la tabla de verdad mediante un **`with ... select`**.
8. Modifique el comparador de 4 bits realizado en el texto para hacerlo genérico. Use 4 como valor por defecto para el parámetro.

CAPÍTULO 7

Circuitos secuenciales. Fundamentos

7.1. Introducción

Los circuitos que se han estudiado hasta ahora tenían como característica fundamental el que las salidas eran una función del valor actual de las entradas. Precisamente por esto se les denomina circuitos combinacionales ya que las salidas son una combinación de las entradas. En este capítulo se estudia otro tipo de circuitos en los que las salidas dependen no sólo del valor actual de las entradas, sino también de los valores pasados. A este tipo de circuitos se les denomina secuenciales, ya que su salida depende de la secuencia de valores que toman las entradas.

Para ilustrar cómo es posible que un circuito dependa del valor anterior de las entradas nada mejor que ver un ejemplo. En la figura 7.1 se muestra un circuito secuencial en el que la salida S se ha conectado a una de las entradas de la puerta OR. En el cronograma de la derecha se puede ver la evolución temporal del circuito. Si se supone que la salida S y la entrada E valen cero inicialmente, como $0 + 0 = 0$ la salida permanecerá a cero mientras la entrada E siga a cero. Cuando E se ponga a uno, la salida S pasará a valer uno después del retardo de la puerta y a partir de ese momento, como $E + 1 = 1$ la salida permanecerá a uno indefinidamente, haga lo que haga la entrada E.

Como puede apreciar, la salida de este circuito depende no sólo del valor actual de la entrada E, sino de la historia pasada de dicha entrada. Así, si la entrada E nunca ha valido uno, la salida S valdrá cero, pero si la entrada E ha tomado alguna vez el valor uno, la salida S será igual a uno, independientemente del valor actual de la entrada E.

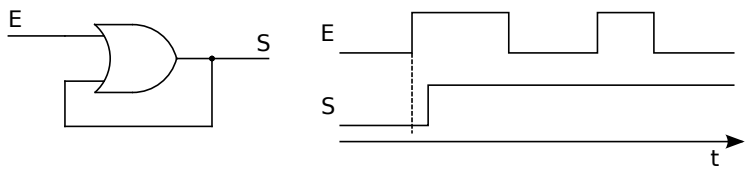


Figura 7.1: Circuito secuencial y su evolución temporal.

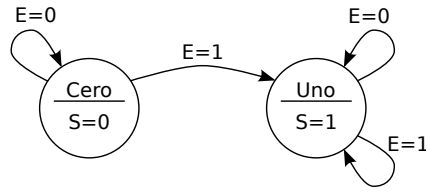


Figura 7.2: Diagrama de estados.

Nótese que este circuito, al igual que el resto de los circuitos secuenciales, tiene memoria, que es lo que le permite reaccionar a una secuencia de entradas. En este caso la memoria no es muy útil, ya que se pone a uno y no hay forma de volver a ponerla a cero; pero en seguida veremos circuitos en los que es posible memorizar cualquier valor. En cualquier caso hay que destacar que la memoria se consigue mediante la **realimentación** de las salidas. En este circuito se ha realimentado la salida S a una de las entradas de la puerta OR.

7.2. Conceptos básicos

Estado

Un circuito secuencial está en cada momento en un estado u otro en función del valor almacenado en sus elementos de memoria. El paso de un estado a otro está gobernado por el valor de las entradas del circuito. Además las salidas también dependen del estado del circuito.

Por ejemplo, en el circuito de la figura 7.1 existen dos estados posibles, uno al que denominaremos `Cero` que es el estado inicial del circuito y otro, al que denominaremos `Uno` que es el estado al que se llega cuando la entrada E se pone a uno.

Diagrama de estados

El comportamiento de los circuitos secuenciales puede representarse gráficamente mediante un diagrama de estados. En estos diagramas cada estado se representa mediante un círculo y las transiciones entre estados mediante flechas. Las transiciones se activan cuando las entradas del circuito toman un determinado valor, el cual se marca en la flecha. Por ejemplo, en la figura 7.2 se muestra el diagrama de estados del circuito de la figura 7.1. Normalmente el valor de las salidas del circuito se asocian a estados y se representan en el diagrama dentro del círculo. En el ejemplo de la figura 7.2 cuando el circuito está en el estado `Cero` la salida vale cero.

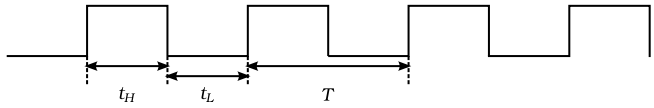


Figura 7.3: Señal de reloj.

Sincronización

El circuito mostrado en la figura 7.1 cambia de estado en cuanto lo hace su entrada. A este tipo de circuitos se les denomina **asíncronos**.

Cuando existen varios elementos de memoria, es preciso que todos cambien de estado a la vez. Se dice entonces que el circuito es **síncrono**. En este caso existe una señal, denominada reloj, que es la que marca los instantes de tiempo en los que se puede cambiar de estado; es decir, es la que sincroniza el sistema.

Reloj

El reloj no es más que una señal cuadrada periódica, tal como se muestra en la figura 7.3. Esta señal está definida por su periodo (o frecuencia) y por su factor de servicio (*duty cycle*), definido como el tiempo que permanece el reloj a uno entre el periodo total, es decir, $FS = t_H/T$. En la práctica los relojes tienen un factor de servicio del 50 %.

7.3. Biestables

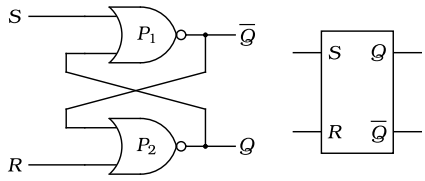
El elemento fundamental de los circuitos secuenciales son los biestables. Como su propio nombre indica, son circuitos con dos estados estables, lo cual les permite almacenar un bit.

Existen en el mercado diversos tipos de biestables, los cuales se clasifican en el cuadro 7.1 en función de su lógica de disparo y de su tipo de sincronismo. De todos ellos los únicos que tienen utilidad práctica son los mostrados en negrita, que son los que vamos a estudiar en este texto.

	Lógica de disparo			
Asíncronos	**R–S**	J–K		
Síncronos por nivel (*latch*)	R–S	J–K	**D**	
Síncronos por flanco (*flip-flop*)	R–S	**J–K**	**D**	**T**

Cuadro 7.1: Clasificación de los biestables.

(a) Circuito y símbolo



(b) Tabla de verdad

S	R	Q_{t+1}	$\overline{Q_{t+1}}$	
1	0	1	0	Puesta a 1 (*Set*)
0	1	0	1	Puesta a 0 (*Reset*)
0	0	Q_t	$\overline{Q_t}$	Mantenimiento
1	1	0	0	Prohibido

Figura 7.4: Biestable R–S con puertas NOR.

7.3.1. Biestable R–S asíncrono

En la figura 7.4 se muestra un biestable R–S construido con puertas NOR. El circuito dispone de dos entradas: S (*Set*) para ponerlo a 1 y R (*Reset*) para ponerlo a cero. El circuito también dispone de dos salidas, las cuales al realimentarse hacen posible que el circuito se comporte como una memoria. Estas salidas Q y $\overline{Q}$ se han denominado así porque en condiciones normales una es la negada de la otra. El funcionamiento del circuito es el siguiente:

- Cuando se activa sólo la entrada S, la salida de la puerta P_1, a la que hemos denominado $\overline{Q}$, se pondrá a cero independientemente del valor de la otra entrada de la puerta. Como $\overline{Q}$ está conectada a la entrada de la puerta P_2, la salida de esta puerta, a la que hemos denominado Q, se pondrá a 1, pues R también vale cero. Este uno ahora se realimenta a la entrada de la puerta P_1, por lo que cuando la entrada S deje de valer uno la salida de la puerta seguirá valiendo cero y el estado del circuito es estable. Nótese que si la entrada S deja de valer uno antes de terminar este ciclo, la salida quedará indefinida, pues no le dará tiempo al biestable a estabilizarse.
- Cuando se activa la sólo la entrada R, la salida Q se pone a cero, independientemente del valor de la otra entrada de la puerta P_2. Como Q está conectada a la entrada de la puerta P_2, la salida de esta puerta se pondrá a 1, pues S también vale cero, pasando por tanto la salida $\overline{Q}$ a valer uno. Este uno se realimenta a la otra entrada de la puerta P_2, por lo que cuando la entrada R deje de valer uno la salida de la puerta seguirá valiendo cero y el estado del circuito es estable. Al igual que antes, si la entrada R deja de valer uno antes de terminar este ciclo, la salida quedará indefinida, pues no le dará tiempo al biestable a estabilizarse.
- Cuando ambas entradas R y S están inactivas, las salidas mantienen su valor anterior. Si por ejemplo $Q_t = 0$ y $\overline{Q_t} = 1$ es fácil ver que un instante después ambas salidas seguirán valiendo lo mismo, pues la puerta P_1 tiene en sus entradas dos ceros, con lo que la salida seguirá siendo un uno y la puerta P_2 tiene en sus entradas un uno y un cero, con lo que su salida seguirá siendo cero. De la misma forma si $Q_t = 1$ y $\overline{Q_t} = 0$ un instante después ambas salidas seguirán valiendo lo mismo, pues la puerta P_1 tiene una entrada a uno y la

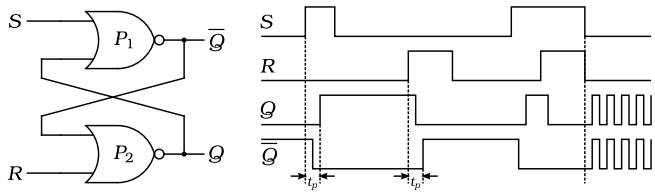


Figura 7.5: Diagrama de tiempos del biestable R–S.

puerta P_2 tiene ambas entradas a cero.

- Cuando ambas entradas valen uno ambas puertas pondrán su salida a cero. No obstante esta es una situación que no tiene sentido, pues el modo de funcionamiento de este circuito consiste en activar la señal *S* cuando queramos poner la salida *Q* a 1 (*Set*) y activar la señal *R* cuando queramos ponerla a cero (*Reset*). Obviamente decirle al circuito a la vez que se ponga a uno y a cero es como decirle a alguien que gire a la derecha y a la izquierda: son órdenes contradictorias y por tanto ha de evitarse. Además, como veremos a continuación el circuito se puede volver inestable al salir de este estado, lo cual es otra razón para evitarlo.

Para expresar este comportamiento de forma sistemática se puede usar un diagrama de estados o una tabla de verdad. En este caso hay que buscar una manera de expresar la realimentación, pues una misma señal es a la vez entrada y salida. Para distinguir las dos facetas de la señal lo que se hace es tener en cuenta que las salidas siempre están retrasadas respecto a las entradas, pues como sabrá las puertas tienen un retardo. Por ello a las señales realimentadas cuando las "vemos" como entradas las denominamos con el subíndice t y cuando las "vemos" como salidas las denominamos con el subíndice $t+1$. Así, en la tabla de verdad del biestable R–S mostrada en la figura 7.4 decimos que las salidas del circuito son Q_{t+1} y $\overline{Q_{t+1}}$ y las entradas *R*, *S*, Q_t y $\overline{Q_t}$.

Respuesta temporal

En la figura 7.5 se muestra la evolución temporal de las señales del biestable R–S. En la figura se ha supuesto que inicialmente la salida *Q* está a cero (y por tanto la $\overline{Q}$ estará a uno). Cuando se activa el *Set*, después del tiempo de propagación de la puerta P_1 baja la señal $\overline{Q}$. Este nuevo valor llegará a la entrada de la puerta P_2 y después del tiempo de propagación de esta puerta la señal *Q* se pondrá a uno. Por tanto, el tiempo de propagación del biestable, al que denominamos t_p será igual a la suma de los retardos de ambas puertas. El valor de t_p es uno de los parámetros fundamentales de un biestable, ya que dos dice el tiempo que tardan en actualizarse sus salidas cuando cambia alguna de sus entradas.

Si se observa el cronograma, cuando sube la señal *R* el proceso es similar al anterior: baja *Q* y esta bajada hace que $\overline{Q}$ suba.

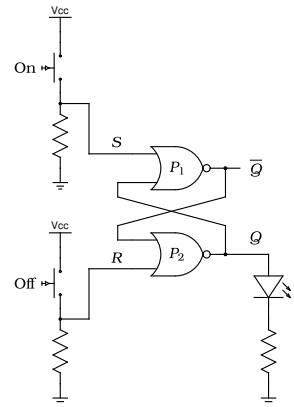


Figura 7.6: Circuito Marcha-Paro con un biestable R–S.

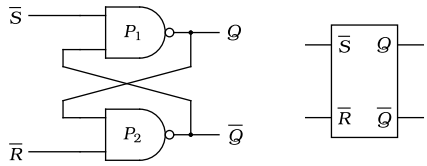
Por último, cuando las dos señales S y R pasan de valer "11" a valer "00" el funcionamiento del circuito es erróneo. En este caso, como ambas salidas están a cero, al ponerse simultáneamente las dos entradas a cero ambas puertas ven dos ceros en sus entradas, por lo que después de su retardo de propagación pondrán a 1 sus salidas. En este momento Q y $\overline{Q}$ pasan a valer 1. Cuando este valor llega a las entradas de las puertas, ambas decidirán que tienen que poner a cero sus salidas, pero eso ocurrirá después del tiempo de propagación. Así volveremos al estado anterior, repitiéndose el proceso indefinidamente.

Normalmente la oscilación anterior o no se dará o terminará tarde o temprano, pues llegará un momento en el que una puerta dé su salida un poco antes que la otra, con lo que "ganará la carrera" y el circuito se quedará en ese valor. Por ejemplo, si en el circuito anterior la puerta P_1 es más rápida que la P_2, cuando se desactiven S y R simultáneamente la salida $\overline{Q}$ se pondrá a uno antes que Q, con lo que le habrá ganado la carrera y el biestable se quedará en el estado $Q = 0$, $\overline{Q} = 1$. Obviamente si la puerta más rápida es la P_2 el resultado será justo el contrario. Esto presenta un grave problema, y es que no podemos saber a ciencia cierta qué ocurrirá con la salida del biestable desactivamos ambas entradas simultáneamente. Es por ello que este estado del biestable se le ha denominado prohibido, pues no está permitido su uso para evitar este tipo de problemas.

Aplicación práctica

La aplicación fundamental de este tipo de biestables es la de permitir controlar una salida a partir de dos señales, una para activarla y otra para desactivarla. Por ejemplo, supongamos que queremos encender un LED cuando pulsemos un botón y apagarla cuando pulsemos otro botón. El circuito para realizarlo es el mostrado en la figura 7.6. Si ambos botones están sin pulsar las entradas R y S del biestable estarán a cero y el LED mantendrá su valor anterior. Cuando se pulse el botón *On* se

(a) Circuito y símbolo

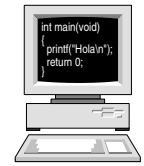


(b) Tabla de verdad

$\overline{S}$	$\overline{R}$	Q_{t+1}	$\overline{Q_{t+1}}$	
0	1	1	0	Puesta a 1 (*Set*)
1	0	0	1	Puesta a 0 (*Reset*)
1	1	Q_t	$\overline{Q_t}$	Mantenimiento
0	0	1	1	Prohibido

Figura 7.7: Biestable R–S con puertas NAND.

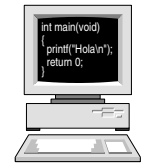
activará la señal *S* y la salida del biestable se pondrá a uno, encendiéndose el LED. Si se suelta el botón, el biestable permanece en el estado de mantenimiento, por lo que el LED se quedará encendido. Para apagarlo se pulsa el botón *Off*, poniendo así la entrada *R* a uno y haciendo que el biestable ponga su salida a cero.



Realice el ejercicio 1

7.3.2. *Biestable R-S asíncrono con puertas NAND*

Se puede también construir un biestable R–S usando puertas NAND. El circuito se muestra en la figura 7.7 y como puede observar el funcionamiento es el mismo que el del biestable R–S con puertas NOR salvo que las entradas de *Set* y *Reset* son activas a nivel bajo y que en estado prohibido las salidas se ponen a uno. En la práctica las puertas NAND son más rápidas y necesitan menos silicio que las NOR, por lo que es prefiere usar este biestable a pesar que que sus entradas activas a nivel bajo sean menos intuitivas de usar.



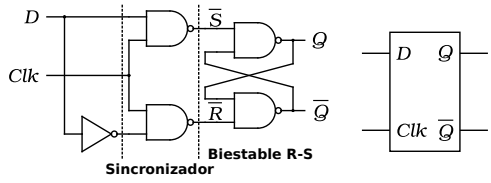
Realice el ejercicio 2

7.3.3. *Biestable síncrono por nivel tipo D*

Los biestables R–S estudiados en la sección anterior modifican sus salidas en cuanto lo hace alguna de sus entradas. No obstante, como se ha dicho en la introducción, cuando existen varios biestables en un circuito es necesario que todos cambien al unísono. Para ello se añade una señal de reloj que controla cuándo el biestable actualiza sus salidas en función de las entradas.

En los biestables síncronos por nivel, también denominados *latch* en inglés, las

(a) Circuito y símbolo

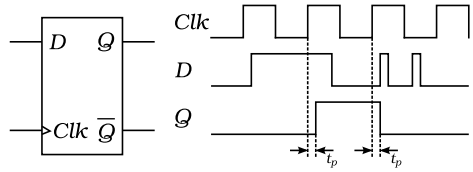


(b) Tabla de verdad

Clk	*D*	Q_{t+1}	$\overline{Q_{t+1}}$	
0	X	Q_t	$\overline{Q_t}$	Manten.
1	0	0	1	Puesta a 0
1	1	1	0	Puesta a 1

Figura 7.8: Biestable D síncrono por nivel.

(a) Símbolo y cronograma



(b) Tabla de verdad

Clk	*D*	Q_{t+1}	$\overline{Q_{t+1}}$	
0	X	Q_t	$\overline{Q_t}$	Manten.
1	X	Q_t	$\overline{Q_t}$	Manten.
↑	0	0	1	Puesta a 0
↑	1	1	0	Puesta a 1

Figura 7.9: Biestable D síncrono por flanco.

salidas sólo pueden cambiar cuando el reloj está activo, permaneciendo inmóviles cuando el reloj está inactivo. De estos biestables el único que tiene interés en la práctica el el tipo D, cuyo circuito se muestra en la figura 7.8.

Como puede apreciar el circuito consta de un biestable R–S con puertas NAND al que se le ha conectado un circuito sincronizador que mediante la señal de reloj *Clk* controla cuándo se puede actualizar la salida del biestable. Así, cuando esta señal vale cero las dos entradas del biestable R–S estarán a uno y por tanto la salida del biestable se mantiene con su valor anterior, independientemente del valor de la entrada *D*. Cuando *Clk* se pone a uno, si *D* vale cero la entrada de la puerta inferior del sincronizador se pondrá a cero, dando un *Reset* al biestable, con lo que éste se pondrá a cero. Si por el contrario la entrada *D* vale uno, será la puerta superior del sincronizador la que se pondrá a cero dando un *Set* al biestable, con lo que su salida se pondrá a uno.

En resumen, este biestable copia el valor de su entrada D en sus salidas mientras la entrada de reloj está activa y mantiene el último valor cuando la entrada de reloj se desactiva. Por tanto este biestable nos permite almacenar un dato y de ahí es precisamente de donde viene su nombre: D de Dato.

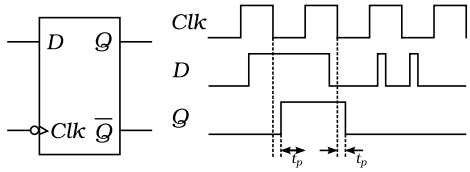
7.3.4. Biestable síncrono por flanco tipo D

El inconveniente de los biestables por nivel es que mientras el reloj está a uno la salida es una copia de la entrada, por lo que si en la entrada se produce algún parpadeo (*glitch*) éste se transferirá a la salida. En los biestables síncronos por flanco el biestable sólo modifica sus salidas cuando se produce un flanco en la señal de reloj. En la figura 7.9 se muestra un *flip-flop* tipo D junto con un cronograma de su funcionamiento y su tabla de verdad.

Las diferencias frente al *latch* tipo D existen obviamente sólo en la forma de sincronización. En primer lugar en el símbolo del biestable se añade un triángulo en la entrada de reloj para indicar que es un biestable disparado por flanco. En segundo lugar en la tabla de verdad puede observar que tanto cuando el reloj vale cero como cuando vale uno la salida del biestable se mantiene a su valor anterior. Sólo cuando se produce un flanco de subida, indicado en la tabla mediante una flecha, el biestable copia en su salida el valor de la entrada.

Por último, en el cronograma se observa que la salida sólo se actualiza en los

(a) Símbolo y cronograma



(b) Tabla de verdad

Clk	*D*	Q_{t+1}	$\overline{Q_{t+1}}$	
0	X	Q_t	$\overline{Q_t}$	Manten.
1	X	Q_t	$\overline{Q_t}$	Manten.
↓	0	0	1	Puesta a 0
↓	1	1	0	Puesta a 1

Figura 7.10: Biestable D síncrono por flanco de bajada.

flancos de subida del reloj, aunque al igual que en el *latch* un tiempo de propagación t_p después.

Realice el ejercicio 3

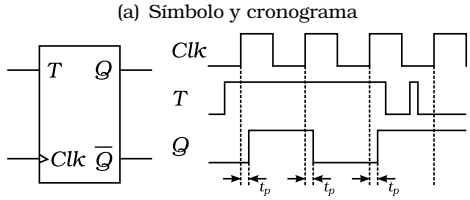
Sincronización con el flanco de bajada

Aunque normalmente se usa el flanco de subida, existen biestables sincronizados por el flanco de bajada. En la figura 7.10 se muestra un *flip-flop* tipo D sincronizado por el flanco de bajada. Como puede observar el funcionamiento es idéntico salvo por el momento en el que se actualiza las salidas. Para indicar que el *flip-flop* está sincronizado por el flanco de bajada en la entrada de reloj se coloca un círculo de negación y en la tabla de verdad se indica el flanco mediante una flecha hacia abajo.

7.3.5. Biestable síncrono por flanco tipo T

En estos biestables la entrada T controla si se invierte la salida o si se mantiene a su valor anterior, tal como se puede observar en la tabla de verdad de la figura 7.11. Aunque el funcionamiento de este tipo de biestables pueda parecer un poco raro, su utilidad queda manifiesta cuando observamos el diagrama de tiempos de la figura. Si lo observa detenidamente verá que cuando $T = 1$ la salida es una onda cuadrada de frecuencia igual a la mitad del reloj de entrada del *flip-flop* T. Así, estos biestables son útiles para realizar circuitos que dividan el reloj a frecuencias más bajas, tal como veremos más adelante.

(a) Símbolo y cronograma

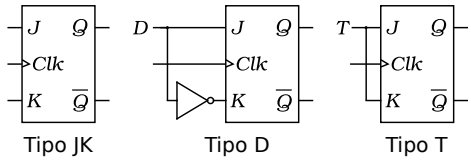


(b) Tabla de verdad

Clk	*T*	Q_{t+1}	$\overline{Q_{t+1}}$	
0	X	Q_t	$\overline{Q_t}$	Manten.
1	X	Q_t	$\overline{Q_t}$	Manten.
↑	0	Q_t	$\overline{Q_t}$	Manten.
↑	1	$\overline{Q_t}$	Q_t	Inversión

Figura 7.11: Biestable T síncrono por flanco.

(a) Símbolo y cronograma



(b) Tabla de verdad

Clk	*J*	*K*	Q_{t+1}	$\overline{Q_{t+1}}$	
0	X	X	Q_t	$\overline{Q_t}$	Manten.
1	X	X	Q_t	$\overline{Q_t}$	Manten.
↑	0	0	Q_t	$\overline{Q_t}$	Manten.
↑	1	0	1	0	Puesta a 1
↑	0	1	0	1	Puesta a 0
↑	1	1	$\overline{Q_t}$	Q_t	Inversión

Figura 7.12: Biestable J–K síncrono por flanco.

7.3.6. *Biestable J-K síncrono por flanco*

Este biestable recibe su nombre en honor a Jack Kilby, su inventor. En la figura 7.12 se muestra su símbolo y su tabla de verdad. Si observa la tabla verá que es similar a la del biestable R–S salvo que cuando ambas entradas están activas a la vez, en lugar de entrar en un estado prohibido el *flip-flop* invierte el valor de la salida.

La principal ventaja de este biestable es el englobar el funcionamiento de los tres anteriores: R–S, D y T; es decir, puede considerarse como un *flip-flop* universal. El comportamiento como R–S se obtiene usando la entrada J como S y la K como R, pues como puede ver en la tabla de verdad la J pone a uno el *flip-flop* y la K a cero. Si se hace que K sea el inverso de J, el biestable funciona como un tipo D, pues cuando D sea cero se hace una puesta a cero y cuando sea uno se hace una puesta a uno. Por último, si unimos la J con la K tenemos un T, pues cuando ambas entradas son cero la salida se mantiene y cuando son uno se invierte.

7.3.7. *Entradas asíncronas*

Cuando se alimenta por primera vez un biestable, el valor que aparece en la salida es aleatorio. Sin embargo los circuitos digitales han de ser deterministas, por lo que es preciso que al arrancar un circuito secuencial todos los biestables se pongan a un valor inicial concreto. Para ello los biestables síncronos suelen tener dos entradas adicionales que permiten inicializarlos a uno o a cero, o al menos una para inicializarlos a cero. Estas entradas funcionan de forma independiente al reloj, de ahí su nombre, y se suelen denominar *Preset* para inicializar a uno y *Clear* para inicializar a cero. Además también es frecuente que dichas entradas sean activas a nivel bajo.

En la figura 7.13 se muestra un *flip-flop* tipo D con entradas asíncronas, así como un cronograma con su funcionamiento. Como puede observar, la activación de la entrada *Pr* (*Preset)* hace que la salida se ponga a uno, como siempre después de un retardo de propagación. De la misma forma la activación de *Cl* (*Clear*) pone la salida a cero. Nótese que ambas entradas actúan inmediatamente, sin necesidad

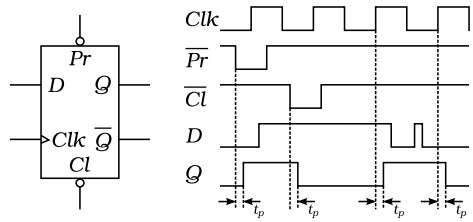


Figura 7.13: *Flip-flop* tipo D con *Preset* y *Clear*.

de que ocurra un flanco en el reloj. Por último indicar que aunque en la figura se han supuesto todos los retardos iguales por simplificar, en la práctica los retardos de propagación desde las entradas asíncronas a las salidas suelen ser distintos de los retardos desde el reloj a las salidas.

Nomenclatura en las señales activas a nivel bajo

En la figura 7.13 se ha mostrado como nombre de la señal *Preset* en el símbolo *Pr* y en el diagrama $\overline{Pr}$. Aunque parezca una errata no lo es. En el símbolo ya estamos diciendo que la señal *Pr* es activa a nivel bajo al poner círculo de negación en la patilla; por lo que no es preciso hacerlo en el nombre de la señal. Si se fija con la salida $\overline{Q}$ se ha hecho lo contrario, en lugar de poner el círculo en la patilla se ha puesto la barra sobre el nombre. En cambio, en el cronograma la única manera que tenemos de decir que una señal es activa a nivel bajo es mediante la barra de negación, que es lo que hemos hecho en la figura.

7.3.8. Parámetros tecnológicos

Aunque los biestables los trataremos siempre a nivel lógico, al igual que las puertas están compuestos por circuitos que necesitan que se respeten ciertos tiempos y ciertos niveles de tensión. En cuanto a estos últimos, los requisitos son los mismos que los de las puertas lógicas, que ya se expusieron en la sección 1.2. En cuanto a los primeros, existen varias restricciones que pasamos a enumerar a continuación:

Ancho mínimo de los pulsos

Para que un biestable asíncrono reconozca una señal de entrada, esta ha de tener un ancho de pulso mayor que un mínimo, tal como se dijo en la sección 7.3.1. Con los relojes de los biestables síncronos y con sus entradas asíncronas de inicialización ocurre lo mismo, han de tener un ancho de pulso suficiente para que al biestable le de tiempo a estabilizarse. Este parámetro se especifica por los fabricantes en las hojas de características de sus circuitos y se suele denominar t_W.

Tiempos de establecimiento y de mantenimiento

En los biestables síncronos por flanco las entradas (D, T, J o K) no pueden cambiar en el mismo tiempo en el que se produce un flanco de reloj. Concretamente existe un tiempo de establecimiento (*setup* en ingles) antes del flanco del reloj y un tiempo de mantenimiento (*hold* en ingles) en el que las entradas han de permanecer constantes. Si se viola esta temporización el estado final del biestable es impredecible.

Estos tiempos también vienen especificados en las hojas de carácterísticas de los circuitos, denominandose normalmente t_S (*setup*) y t_H (*hold*.

Metaestabiliad

Si la señal de entrada al *flip-flop* (D, T, J o K) viola los tiempos de mantenimiento o establecimiento, el *flip-flop* puede entrar en un estado metaestable, el cual consiste en que la salida permanece unos instantes a un valor intermedio entre cero y uno, normalmente alrededor de la mitad de la tensión de alimentación, para luego caer aleatoriamente a cero o a uno. El problema adicional a la aleatoriedad del valor final es que mientras que la salida permanece a un valor intermedio, el resto de circuitos conectados a esta salida interpretarán esta señal a su manera, unos a cero y otros a uno; convirtiéndose entonces el circuito digital en una ruleta.

Tiempos de propagación

Al igual que las puertas lógicas, los biestables tardan un tiempo en actualizar sus salidas después de los cambios en las entradas. En el caso de los biestables síncronos por nivel se especifica un tiempo de propagación desde el cambio en las entradas de datos hasta la salida y otro desde el cambio del reloj hasta la salida. En el caso de los *flip-flop* sólo tiene sentido especificar el tiempo de propagación desde que se produce el flanco hasta que cambia la salida. Por último, si el biestable dispone de entradas asíncronas de inicialización, existirá también un tiempo de propagación desde el cambio de estas entradas hasta el cambio de las salidas.

7.4. Ejercicios

1. En el circuito de la figura 7.6 no se han dado valores a las resistencias. Calcúlelos teniendo en cuenta los siguientes datos:
   - Para que el LED se encienda es necesario que pase por él una corriente de 5 mA. Suponga que la caída de tensión en el LED cuando éste conduce es de $V_f = 1,4\text{V}$.
   - Los circuitos se alimentan a $V_{cc} = 3,3\text{V}$.
   - Diseñe las resistencias de los pulsadores para que cuando éstos se pulsen pase por ellos una corriente de 0,1 mA.
   - Suponga que las puertas lógicas tienen las siguientes características:

- $i_{IH} = 1\mu A$
- $V_{OL} = 0,1V$
- $V_{OH} = 3,1V$

2. Modifique el circuito de la figura 7.6 para usar un biestable con puertas NAND.
3. En la figura 7.9 se muestra un cronograma con el funcionamiento de un *flip-flop* tipo D. Repita este cronograma para representar en funcionamiento de un *latch* tipo D sometido a estas mismas entradas.

CAPÍTULO 8

Máquinas de estados finitos

8.1. Introducción

Las máquinas de estados finitos, frecuentemente denominadas simplemente máquinas de estados o también autómatas finitos; nos permiten diseñar circuitos secuenciales complejos, capaces de "tomar decisiones" en función de un estado actual del circuito y del valor de sus entradas. En la práctica se utilizan normalmente como circuitos de control.

8.2. Nomenclatura

En una máquina de estados tenemos:

- X entradas.
- Y salidas.
- Q variables de estado, almacenadas en una memoria.
- δ función de transición entre estados.
- λ función de salida.

Dependiendo de si las salidas dependen sólo del estado o también de las entradas tenemos dos tipos de máquinas de estados:

Máquina de Moore

Este tipo de máquinas reciben el nombre de su inventor: Edward Forrest Moore. En ellas las salidas sólo dependen de las variables de estado. Así las funciones δ y λ se definen como:

$$\begin{aligned} Q_{t+1} &= \delta(X, Q_t) \\ Y &= \lambda(Q_t) \end{aligned}$$

En donde la función δ está sincronizada normalmente por un reloj activo por flanco. Una representación gráfica de este tipo de máquinas puede observarse en la figura 8.1.

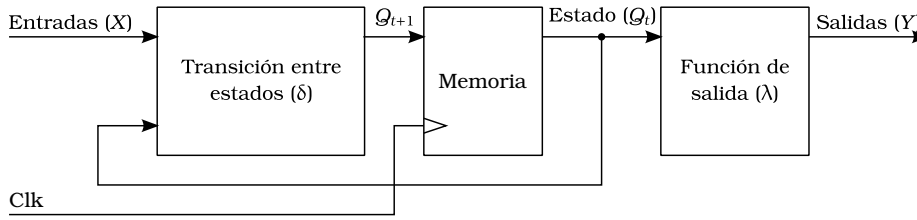


Figura 8.1: Diagrama de bloques de una máquina de Moore.

Máquina de Mealy

Al igual que la máquina de Moore, este tipo de máquinas se denominan así en honor a su inventor, George H. Mealy. A diferencia de las máquinas de Moore, las salidas dependen del estado y de las entradas; es decir, las funciones δ y λ son:

$$\begin{aligned} Q_{t+1} &= \delta(X, Q_t) \\ Y &= \lambda(X, Q_t) \end{aligned}$$

Al igual que en la máquina de Moore, la función δ está sincronizada normalmente por un reloj activo por flanco. En la figura 8.1 se muestra un diagrama de bloques de este tipo de máquinas, donde se aprecia claramente que la única diferencia es que ahora las entradas influyen directamente en la función de salida λ.

El inconveniente de este tipo de máquinas es que al depender la salida directamente de las entradas su temporización es más difícil de analizar. Además, si aparece algún *glitch* en las entradas, éste se propaga hasta las salidas, mientras que en las máquinas de Moore, al depender las salidas sólo de las variables de estado esto no ocurre. La principal ventaja de las máquinas de Mealy es que en general necesitan menos estados para realizar la misma funcionalidad; pero esto no compensa sus inconvenientes, por lo que en la práctica se prefiere usar las máquinas de Moore.

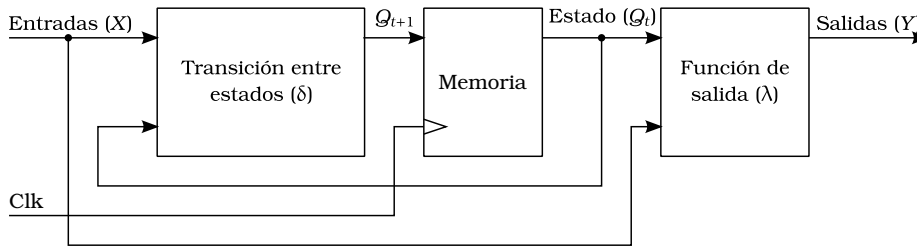


Figura 8.2: Diagrama de bloques de una máquina de Mealy.

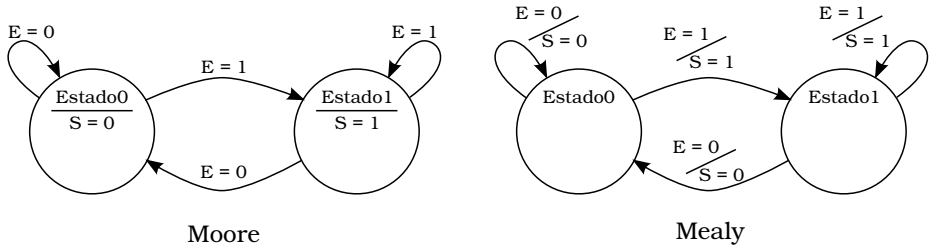


Figura 8.3: Diagramas de estados para máquinas Moore y Mealy.

8.2.1. Representación gráfica

La representación gráfica mediante diagramas de estados es distinta según se trate de una máquina de Moore o de Mealy. En ambas los estados se representan mediante círculos y las transiciones mediante flechas, en las cuales se representa el valor de las entradas que dispara dicha transición. La diferencia está en las salidas. Como en las máquinas de Moore las salidas sólo dependen del estado, se dibujan dentro de los círculos, tal como se puede apreciar en la figura 8.3. En cambio en las máquinas de Mealy el valor de las salidas se indica en las transiciones.

8.3. Diseño de máquinas de estados

El proceso de diseño de una máquina de estados consiste en obtener las funciones δ y λ a partir de las especificaciones del circuito. El proceso es algo tedioso, aunque afortunadamente los lenguajes de descripción de hardware nos permiten describirlas y hacer la síntesis automáticamente. Incluso hay paquetes de *software* que permiten dibujar un diagrama de estados y a partir de éste obtienen el código en VHDL o Verilog. No obstante, antes de ver cómo se describe una máquina de estados en VHDL vamos a estudiar cómo diseñarla "a mano".

El proceso de diseño consta de las fases que se indican en la figura 8.4. La parte creativa y un poco complicada a veces es el paso de la especificación del sistema al diagrama de estados. Una vez realizado este paso, el resto es algo sistemático que

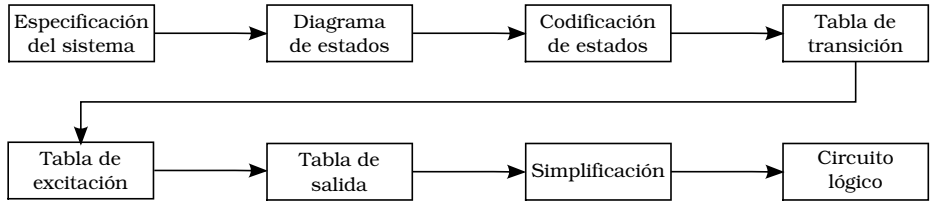


Figura 8.4: Proceso de diseño de una máquina de estados.

incluso puede automatizarse mediante un ordenador como acabamos de decir.

Para ilustrar el proceso de diseño vamos a realizar como ejemplo un detector de flanco de subida. El circuito ha de generar un pulso en su salida cuando se detecte un paso de cero a uno en la señal de entrada. En primer lugar diseñaremos el circuito usando una máquina de Moore y a continuación repetiremos el proceso usando una máquina de Mealy.

8.3.1. Detector de flanco mediante una máquina de Moore

El diagrama de estados del detector de flanco se muestra en la figura 8.5. El circuito arranca después del reset en el estado Esp1, donde espera a que la entrada pase a uno. Cuando esto ocurra pasamos al estado Pulso, en donde se activa la salida para indicar que se ha detectado un flanco. En este estado estaremos sólo durante un ciclo, volviendo al estado Esp1 si la señal de entrada vale cero en el siguiente flanco de reloj o yendo al estado Esp0 a esperar a que la entrada se ponga a cero.

Codificación de estados

El siguiente paso en el proceso de diseño consiste en la codificación de los estados en binario. En el diagrama de estados a cada estado se le ha dado un nombre simbólico que nos recuerda lo que está haciendo el circuito en cada estado. Por ejemplo en el estado Esp1 está esperando a que la entrada se ponga a uno. Estos nombres simbólicos han de codificarse como unos y ceros antes de poder introducirse en un circuito digital. Existen numerosas formas de codificar estos estados. En primer lugar podemos asignarles un número binario, asignando el 00 al estado Esp1, el 01 al estado Pulso y el 10 al estado Esp0. A esta forma de codificación se le denomina codificación **binaria**. Otra alternativa es usar un bit para cada estado. Así el estado Esp1 se codificaría como 001, el estado Pulso como 010 y el estado Esp0 como 100. A este último método de codificación se le denomina *one hot* y aunque necesita más biestables para almacenar el estado presenta la ventaja de necesitar menos lógica para calcular el estado siguiente y la salida (funciones δ y λ); por lo que son más rápidas. Además es frecuente su uso cuando se implantan máquinas de muchos estados en una FPGA, ya que en éstas se implantan mejor circuitos con muchos *flip-flops* que circuitos con funciones lógicas complejas.

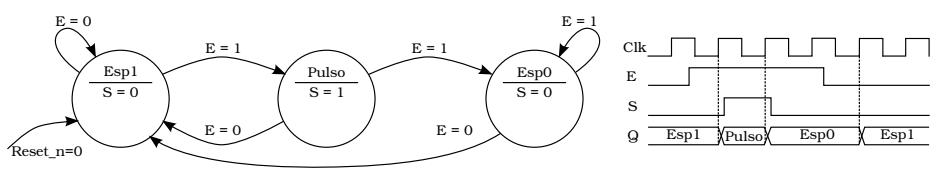


Figura 8.5: Diagrama de estados y cronograma del detector de flanco por Moore.

Estado$_t$	Q_1^t	Q_0^t	E	Q_1^{t+1}	Q_0^{t+1}	Estado$_{t+1}$
Esp1	0	0	0	0	0	Esp1
Esp1	0	0	1	0	1	Pulso
Pulso	0	1	0	0	0	Esp1
Pulso	0	1	1	1	0	Esp0
Esp0	1	0	0	0	0	Esp1
Esp0	1	0	1	1	0	Esp0

Cuadro 8.1: Tabla de transición

Estado$_t$	Q_1^t	Q_0^t	E	Q_1^{t+1}	Q_0^{t+1}	Estado$_{t+1}$	D_1	D_0
Esp1	0	0	0	0	0	Esp1	0	0
Esp1	0	0	1	0	1	Pulso	0	1
Pulso	0	1	0	0	0	Esp1	0	0
Pulso	0	1	1	1	0	Esp0	1	0
Esp0	1	0	0	0	0	Esp1	0	0
Esp0	1	0	1	1	0	Esp0	1	0

Cuadro 8.2: Tabla de excitación

Tabla de transición

A partir del diagrama de estados podemos obtener la tabla de transición de estados, que contiene la misma información que el diagrama pero de forma tabular. Dicha tabla contiene a su izquierda el estado actual y el valor de las entradas de la máquina y a su derecha el estado siguiente. El estado es conveniente representarlo de forma simbólica y codificado. La primera representación es para facilitarnos la lectura de la tabla a nosotros, mientras que la segunda es necesaria para la obtención del circuito lógico.

En la tabla 8.1 se muestra la tabla de transición del detector de flanco. Como puede observar los estados se han codificado mediante codificación binaria. Nótese que al necesitar dos bits para representar el estado, a éstos se les ha denominado Q_1 y Q_0, usando el superíndice t para indicar el estado actual y el superíndice $t+1$ para indicar el estado siguiente.

Tabla de excitación

A partir de la tabla anterior se crea la tabla de excitación, en la que se indica el valor que han de tener las entradas de los *flip-flops* para que se produzca la transición deseada del estado. Por tanto, antes de crear esta tabla es necesario decidir qué tipo de *flip-flop* se va a usar para representar el estado. Lo más fácil es usar *flip-flop* tipo D, ya que en este tipo de *flip-flop* si queremos que la salida se ponga a 0 o 1 basta con poner su entrada a dicho valor.

En la tabla 8.2 se muestra la tabla de excitación, que como puede observar es una copia de la tabla de transición a la que se le han añadido las columnas D_1 y D_0

en las que se especifica el valor que han de tener las entradas de los *flip-flops* para que el estado siguiente ($t+1$) sea el especificado. Al usarse *flip-flops* tipo D estas columnas coinciden con las correspondientes al estado siguiente Q_1^{t+1} y Q_0^{t+1}.

Tabla de salida

Una vez creada la tabla de excitación, que nos permitirá obtener la función δ, es necesario crear la tabla de salida, a partir de la que se obtendrá la función λ. Como en la máquina de Moore las salidas dependen sólo del estado actual, en esta tabla se especifica a la izquierda el estado junto con su codificación y a la derecha el valor de la salida correspondiente al estado; tal como se muestra en la tabla 8.3.

Simplificación

En la tabla de excitación tenemos ya todos los datos que necesitamos para obtener las ecuaciones de la función δ. Por un lado las entradas son Q_1^t, Q_0^t y E y las salidas son D_1 y D_0. Por tanto podemos escribir los diagramas de Karnaugh para ambas funciones, los cuales se muestran en la figura 8.6. A partir de estos diagramas se obtienen las ecuaciones lógicas de las entradas de los *flip-flops* encargados de almacenar el estado, las cuales son:

$$\begin{aligned} D_1 &= Q_1 \cdot E + Q_0 \cdot E \\ D_0 &= \overline{Q_1} \cdot \overline{Q_0} \cdot E \end{aligned}$$

De la misma forma, a partir de la tabla de salida (8.3) se obtiene su diagrama de Karnaugh (figura 8.6) y a partir de él la ecuación de la salida:

$$S = Q_0$$

Circuito lógico

El último paso es el diseño del circuito lógico a partir de las ecuaciones obtenidas en el paso anterior. El circuito obtenido en este ejemplo se muestra en la figura 8.7. Nótese que se han usado dos *flip-flops* para almacenar el estado Q_1 y Q_0. Ambos *flip-flops* se han conectado a la señal de reloj (*CLK*) para sincronizarlos y a la de reset ($\overline{Reset}$) para llevarlos al estado inicial, que recuerde que se ha codificado como 00. Como la entrada $\overline{PRE}$ no es necesaria en ambos *flip-flops* se ha conectado a uno

Estado$_t$	Q_1^t	Q_0^t	S
Esp1	0	0	0
Pulso	0	1	1
Esp0	1	0	0

Cuadro 8.3: Tabla de salida

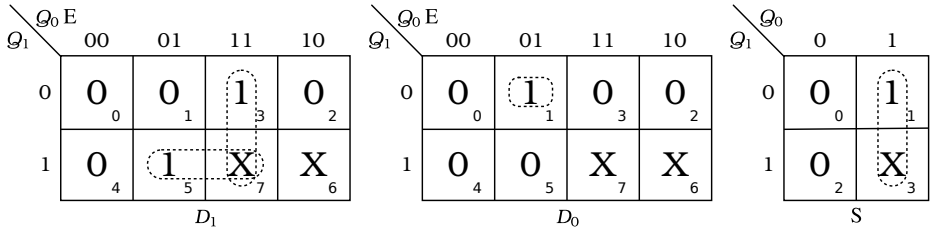
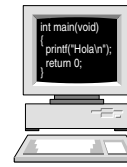


Figura 8.6: Diagramas de Karnaugh del detector de flanco por Moore.

para desactivarla. Por último se han implantado las ecuaciones de las entradas D_1 y D_0 obtenidas en el apartado anterior, así como la de la salida S.



Realice el ejercicio 1

8.3.2. *Detector de flanco mediante una máquina de Mealy*

El proceso de diseño para el detector de flanco mediante una máquina de Mealy es similar al seguido para la máquina de Moore. La única diferencia radica en la forma de expresar el valor de las salidas en el diagrama de estados y en que la tabla de salidas está integrada junto con la tabla de excitación, ya que éstas dependen tanto del estado como de las entradas de la máquina.

En la figura 8.8 se muestra el diagrama de estados y el cronograma del detector de flanco implantado mediante una máquina de Mealy. En este caso el estado representa el valor que tenía la señal de entrada en el flanco de reloj anterior. Así el estado Cero indica que la entrada en el último flanco de reloj valía cero, por lo que si

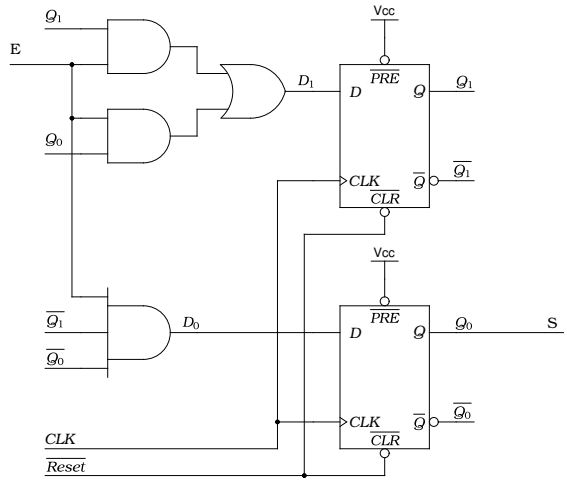


Figura 8.7: Circuito lógico del detector de flanco por Moore.

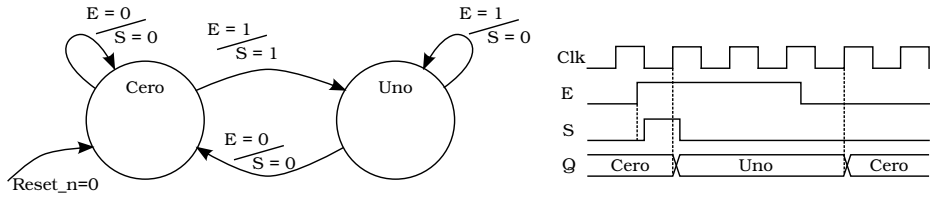


Figura 8.8: Diagrama de estados y cronograma del detector de flanco por Mealy.

Estado$_t$	Q^t	E	Q^{t+1}	Estado$_{t+1}$	S
Cero	0	0	0	Cero	0
Cero	0	1	1	Uno	1
Uno	1	0	0	Cero	0
Uno	1	1	1	Uno	0

Cuadro 8.4: Tabla de transición y salidas

la entrada sigue valiendo cero seguiremos en el mismo estado y la salida valdrá cero, pues no hay ningún flanco. Si por el contrario la entrada vale uno en este estado, se realizará una transición en el siguiente flanco de reloj al estado Uno y además se pondrá a uno la salida S para indicar que se ha producido un flanco de subida en la señal de entrada. Según se aprecia en el cronograma de la figura 8.8, el pulso de salida dura ahora el tiempo desde que la señal de entrada se pone a uno hasta que se realiza el cambio de estado en el siguiente flanco de reloj. Es decir, la salida está ahora asociada a la transición entre el estado Cero y Uno, no a la permanencia en un estado determinado como ocurre en las máquinas de Moore. Cuando estemos en el estado Uno, si la entrada sigue valiendo 1 permanecemos en dicho estado y si vale cero pasamos al estado Cero. En ambos casos la salida valdrá cero para indicar que no hay flancos de subida en la señal de entrada.

Codificación de estados

En este caso vamos a usar también la codificación binaria. Al tener sólo dos estados asignaremos 0 al estado Cero y 1 al estado Uno.

Tabla de transición y salidas

En las máquinas de Mealy las salidas dependen del estado y de las entradas. Por ello es conveniente incluir en la tabla de transición la tabla de las salidas. Se podrían realizar dos tablas separadas, pero por simplificar se prefiere normalmente realizar sólo una. En la tabla 8.4 se muestra la tabla de transición y salidas para el circuito.

Estado$_t$	Q^t	E	Q^{t+1}	Estado$_{t+1}$	J	K	S
Cero	0	0	0	Cero	0	X	0
Cero	0	1	1	Uno	1	X	1
Uno	1	0	0	Cero	X	1	0
Uno	1	1	1	Uno	X	0	0

Cuadro 8.5: Tabla de excitación y salidas

Tabla de excitación

Al igual que con la máquina de Moore, a partir de la tabla de transición de estados es necesario crear la tabla de excitación de los *flip-flops*. En este caso vamos a usar un *flip-flop* tipo J-K para almacenar el bit de estado Q. En este caso la obtención de la tabla de excitación es un poco más complicada que con los *flip-flops* tipo D, ya que en este tipo *flip-flops* el estado final que queremos conseguir depende del estado de partida. En la tabla siguiente se muestra la también llamada tabla de excitación del *flip-flop* J-K:

Q_t	Q_{t+1}	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

En esta tabla se definen los valores que han de tener las entradas J y K para que se produzca la transición deseada en el *flip-flop*. En la primera fila de la tabla se puede ver que si el estado anterior del *flip-flop* es cero y queremos que siga a cero, la entrada J ha de ser 0 y la K da igual lo que valga. Esto es así porque si J y K valen 0, el *flip-flop* mantiene su valor anterior y si J = 0 y K = 1 la salida del *flip-flop* se pone a cero, con lo que se conseguirá de la misma forma que $Q_{t+1} = 0$. En la segunda fila se puede ver que si Q_t es cero y queremos que Q_{t+1} sea uno tenemos dos formas de conseguirlo: o bien hacemos que J y K valgan 1, con lo que el *flip-flop* invierte su salida, o bien hacemos que J sea 1 y K sea 0, con lo que el *flip-flop* pone su salida a 1. El resto de las filas de la tabla se obtienen de la misma forma.

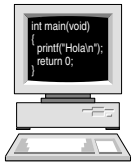
A partir de esta tabla de excitación del *flip-flop* J-K se obtiene la tabla de excitación de la máquina de estados, la cual se muestra en la tabla 8.5. Para la obtención de esta tabla basta con observar en la tabla de transición el valor de Q_t y Q_{t+1} y elegir los valores correspondientes de la tabla de excitación del *flip-flop* J-K, copiándolos a sus columnas correspondientes.

Simplificación

A partir de la tabla de excitación se obtienen los diagramas de Karnaugh de la figura 8.9; y a partir de éstos las ecuaciones de las entradas del *flip-flop* y de la

salida:

$$\begin{aligned} J &= E \\ K &= \overline{E} \\ S &= \overline{Q} \cdot E \end{aligned}$$



Realice el ejercicio 2

Circuito lógico

A partir de las ecuaciones anteriores se obtiene el circuito lógico, mostrado en la figura 8.9

8.4. Descripción en VHDL

La descripción de máquinas de estado en VHDL necesita sentencias más potentes de las que hemos estudiado hasta ahora. Por tanto, antes de pasar a describir una máquina de estados en VHDL es necesario introducir estas nuevas sentencias.

8.4.1. Los **process**

Las sentencias VHDL que hemos estudiado hasta ahora son sentencias concurrentes. Cada una de ellas describe un elemento *hardware* y por ello podemos pensar que se "ejecutan" en paralelo. Además de estas sentencias existen otras que permiten describir el comportamiento de un componente *hardware* de forma secuencial, aunque para integrarlas con el resto de sentencias han de agruparse dentro de un **process**. Este **process** es equivalente a una sola sentencia concurrente que se "ejecuta" en paralelo con el resto de sentencias.

La sintaxis del **process** es la siguiente:

```
[Etiqueta:] process (señal_1, señal_2,  señal_n)
  [declaraciones]
begin
  sentencia_secuencial;
  sentencia_secuencial;
```

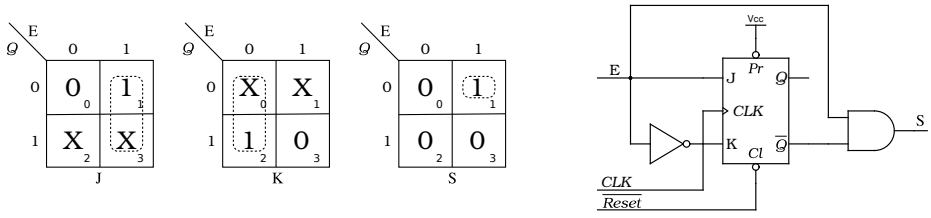


Figura 8.9: Diagramas de Karnaugh y circuito lógico del detector de flanco por Mealy.

```
  sentencia_secuencial;
end process [Etiqueta];
```

Lo único destacable de esta sintaxis es la lista de sensibilidad, que es una serie de señales que se escriben entre paréntesis justo después de la palabra clave **process**. Esta lista de señales ha de contener todas las señales que se leen en el **process** ya que dicho **process** solo se "ejecutará" cuando cambie alguna de las señales indicadas en la lista de sensibilidad. Esta particularidad se usa para optimizar la simulación, ya que cuando el simulador detecta un **process** en el que no ha cambiado ninguna de sus señales de la lista de sensibilidad no lo evalúa y así se ahorra tiempo. El problema ocurre cuando el sintetizador genera el circuito, pues éste estará siempre activo y por tanto los resultados de simulación no coincidirán con los obtenidos en el *hardware* real, lo cual puede ser catastrófico si por ello no detectamos un error hasta tener el chip fabricado.

Por ejemplo, para especificar una puerta AND usando un **process** podemos hacer:

```
process (a, b)
begin
  s <= a and b;
end process;
```

Ya se que es un ejemplo poco útil, pero sirve para indicar dos cosas: que han de escribirse ambas entradas `a` y `b` en la lista de sensibilidad (pero no la salida) y que el operador de asignación es el mismo tanto dentro como fuera de un **process**.

Si en el ejemplo anterior escribimos:

```
process (a)
begin
  s <= a and b;
end process;
```

El sintetizador genera exactamente el mismo circuito, pero el simulador sólo actualizará la salida de la puerta cuando cambie la señal `a`, pero no cuando cambie la `b`, lo cual generará diferencias entre la simulación y la realidad.

8.4.2. La sentencia **if**

la sentencia **if** es similar a la de los lenguajes de programación: permite evaluar una condición lógica y "ejecutar" un código si ésta es cierta. Además existe también la clausula **else** que permite "ejecutar" código si la secuencia es falsa. Por último, al igual que en los lenguajes de programación, también podemos encadenar varios **else if**, aunque en este caso la sintaxis es un poco caprichosa, pues se usa la clausula **elsif** en lugar de un **else** seguido de un **if** como en C.

La sintaxis es la siguiente:

```
if condición then
```

```
  sentencia;
  sentencia;
[elsif condición then
  sentencia;
  sentencia; ]
[else
  sentencia;
  sentencia; ]
end if;
```

Como puede apreciar, tanto el **elsif** como el **else** son opcionales.

Por ejemplo, si queremos matar pulgas a cañonazos e implantar una puerta AND con una sentencia **if** podemos hacer:

```
process (a, b)
begin
  if a = '1' and b = '1' then
    s <= '1';
  else
    s <= '0';
  end if;
end process;
```

Observe en el ejemplo que el **if** se ha escrito dentro de un **process**, ya que es una sentencia secuencial.

Una de las utilidades prácticas del **if** es la de implementar circuitos síncronos. Para ello en la condición se detecta el flanco de reloj y se ejecutan las sentencias que queramos sincronizar dentro del if. Por ejemplo, para implementar un biestable sincronizado por flanco tipo D podemos hacer:

```
process (clk, d)
begin
  if clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```

En este código hay otro elemento nuevo: el atributo de una señal, cuya sintaxis es:

```
señal'atributo
```

Estos atributos permiten obtener información adicional de una señal. Por ejemplo el atributo `event` es verdadero cuando ocurre un flanco en la señal sobre la que se aplica o falso en caso contrario. Así, para detectar un flanco en la señal `clk` basta con escribir: `clk'event`. Si además queremos detectar un flanco de subida se comprueba que además después del flanco la señal valga uno, que es lo que se hace en la condición del **if**.

Resumiendo, en el **if** anterior en la salida q se copia el valor de la entrada d sólo si hay un flanco de reloj, que como recordará es precisamente lo que hace un *flip-flop* tipo D.

Si queremos añadir al *flip-flop* un reset asíncrono activo a nivel bajo, basta con ampliar el **if** anterior para tenerlo en cuenta:

```
process (reset_n, clk, d)
begin
  if reset_n = '0' then
    q <= '0';
  elsif clk'event and clk = '1' then
    q <= d;
  end if;
end process;
```

8.4.3. *La sentencia* **case**

Esta sentencia es el equivalente al **with...select** pero para usarlo dentro de un **process**. Su sintaxis es:

```
case expresión is
  when valor =>
    sentencia;
    sentencia;
  when otro_valor =>
    sentencia;
    sentencia;
  when others =>
    sentencia;
    sentencia;
end case;
```

y su funcionamiento es similar al del **with...select**: se compara la óexpresin con los valores valor, otro_valor, etc. Si se encuentra una coincidencia se ejecutan las sentencias que siguen a dicho valor. Si no se encuentra ninguno, se ejecutan las sentencias que siguen al **when others**. Éste último **when** es en realidad opcional, pero como veremos más adelante lo normal es usarlo siempre.

Por último comentar que es posible asociar sentencias a varios valores separándolos por el operador |.

Por ejemplo, un multiplexor se describiría de la siguiente manera con un **case**:

```
process(sel, a0, a1)
begin
  case sel is
    when '0' =>
      s <= a0;
```

```
    when '1' =>
      s <= a1;
    when others =>
      s <= '0'
  end case;
end process;
```

En este caso es muy importante asignar en todos los **when** un valor a la salida s. Si nos dejamos uno en el que la salida no se actualiza, como por ejemplo:

```
process(sel, a0, a1)
begin
  case sel is
    when '0' =>
      s <= a0;
    -- error: se nos ha olvidado especificar qué hacer
    -- cuando sel = '1'. Se generará un latch
  end case;
end process;
```

al no decir qué hacer con la salida s cuando sel vale uno, el sintetizador interpreta que hay que mantener el valor anterior y nos sintetiza un *latch* para almacenar el valor de s. Este *latch* nos perjudica de dos maneras. En primer lugar ocupa lógica innecesaria y en segundo lugar retrasa la señal un ciclo de reloj. Es por tanto muy importante especificar en primer lugar todos los casos posibles de la entrada de selección y en segundo lugar en todos ellos asignar un valor a la (o las) señales. Para evitar que se generen *latch* si se nos olvida un **when** es conveniente usar siempre el **when others** para darle un valor por defecto a la salida. Otra posibilidad es asignar un valor por defecto a la señal antes de entrar en el **case**, de modo que si se nos olvida algún **when** el sintetizador le da el valor por defecto y no genera ningún *latch*:

```
process(sel, a0, a1)
begin
  s <= '0'; -- valor por defecto
  case sel is
    when '0' =>
      s <= a0;
    -- ahora no pasa nada si se nos olvida un posible
    -- valor de sel
  end case;
end process;
```

8.4.4. Ejemplo. Detector de flanco

En el siguiente listado se muestra un detector de flanco de subida implementado mediante una máquina de Moore. En primer lugar destacar que se define un tipo

denominado `t_estado` para almacenar las variables de estado. Este tipo es un tipo enumerativo que sólo puede tomar en este caso los valores `Esp1`, `Pulso` o `Esp0`. Esto permite evitar errores si intentamos asignar un valor arbitrario a las variables de estado. El cómo se codifican estas variables de estado en binario es tarea del sintetizador, el cual realizará dicha asignación para generar un circuito óptimo. Una vez definido se han creado dos señales de tipo `t_estado`: `estado_act` para almacenar el valor actual de las variables de estado (la salida de los *flip-flops* de estado) y `estado_sig` para almacenar el valor que debe tomar el estado en el siguiente flanco (que será la entrada de los *flip-flops*).

La máquina de estados se ha implementado mediante tres **`process`**. El primero especifica los *flip-flop* donde se almacenan las variables de estado. Como puede observar el **`process`** actualiza el estado (`estado_act`) con el estado siguiente (`estado_sig`) cuando se produce un flanco en el reloj. Este process también se encarga de gestionar el reset asíncrono, llevando la máquina a su estado inicial `Esp1`.

El segundo **`process`** describe el circuito combinacional que calcula el siguiente estado en función de las entradas y el estado actual. Dicho cálculo se realiza en un **`process`**, en el que con un **`case`** se selecciona el estado siguiente en función del estado actual. Para ello, dentro del **`when`** de cada estado se verifica si se activa alguna de las condiciones que hacen que cambie el estado y si es así se actualiza el estado siguiente con el nuevo valor. Para simplificar, antes de entrar en el **`case`** se le da al estado siguiente el valor del estado actual y así no es necesario escribir esta asignación en todos los **`when`** cuando no se activa ninguna señal que cambie de estado.

Por último, en el tercer **`process`** se describe la lógica de las salidas. Al igual que en el process anterior, es más cómodo dar un valor inicial a las salidas y así sólo poner los estados en los que las salidas valen uno. Para dejar constancia de que en ciertos estados no hay que dar un valor a la salida distinto del de por defecto se pone como sentencia un **`null`** que no quiere decir nada más que el diseñador ha dicho que no es preciso hacer nada en este caso. Esto evita pensar a otros diseñadores que vean nuestro código, o a nosotros mismos cuando lo veamos dentro de un tiempo, que se nos olvidó especificar el valor de las salidas en un estado.

```
1  -- Detector de flanco mediante una máquina de Moore
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity DetectorFlanco is
7
8    port (
9      e       : in  std_logic;            -- Entrada
10     reset_n : in  std_logic;            -- Inicialización
11     clk     : in  std_logic;            -- Reloj del sistema
12     s       : out std_logic);           -- Salida
```

```
13
14 end DetectorFlanco;
15
16 architecture behavioral of DetectorFlanco is
17   -- Tipo para los estados
18   type   t_estados is (Esp1, Pulso, Esp0);
19   -- Estados actual y siguiente
20   signal estado_act, estado_sig : t_estados;
21 begin  -- behavioral
22
23   VarEstado : process (clk, reset_n, estado_sig)
24   begin
25     if reset_n = '0' then               -- Reset asíncrono
26       estado_act <= Esp1;               -- (activo bajo)
27     elsif clk'event and clk = '1' then  -- Flanco de subida
28       estado_act <= estado_sig;
29     end if;
30   end process VarEstado;
31
32   TransicionEstados : process (estado_act, e)
33   begin
34     -- Por defecto nos quedamos en el estado actual
35     estado_sig <= estado_act;
36     case estado_act is
37       when Esp1 =>
38         if e = '1' then
39           estado_sig <= Pulso;
40         end if;
41       when Pulso =>
42         if e = '1' then
43           estado_sig <= Esp0;
44         else
45           estado_sig <= Esp1;
46         end if;
47       when Esp0 =>
48         if e = '0' then
49           estado_sig <= Esp1;
50         end if;
51       when others =>          -- Si se salta a un estado raro
52         estado_sig <= Esp1;   -- nos vamos al estado inicial
53     end case;
54   end process TransicionEstados;
55
56   Salidas : process (estado_act)
```

```
57  begin
58    -- Por defecto la salida vale cero
59    s <= '0';
60    case estado_act is
61      when Esp1 =>
62        null;
63      when Pulso =>
64        s <= '1';
65      when Esp0 =>
66        null;
67      when others =>
68        null;
69    end case;
70  end process Salidas;
71 end behavioral;
```

8.5. Detector de secuencia

Para terminar el capítulo nada mejor que estudiar un ejemplo completo un poco más complejo que el detector de flanco. El ejemplo usado va a ser una cerradura electrónica que detecte una secuencia de pulsación de dos teclas y abra la cerradura cuando se introduzca la secuencia correcta. El sistema dispondrá de dos pulsadores P1 y P0 y una salida S. La salida se activará cuando se pulse P1, luego P0 y por último otra vez P0. La salida permanecerá activa hasta que se pulse otra tecla.

En primer lugar hay que tener en cuenta que como el reloj del sistema es mucho más rápido que el usuario, una pulsación de una tecla durará muchos ciclos de reloj. Por tanto se necesitan dos estados para detectar una pulsación completa: uno para detectar la pulsación y otro para detectar la liberación. Esto es lo que se ha hecho en el diagrama de estados mostrado en la figura 8.10.[1] Para simplificar el diagrama en las transiciones se muestran tan solo el valor de las dos entradas p1 y p0. Para que no haya lugar a dudas del orden, en la esquina superior izquierda del diagrama se ha escrito $\frac{p1p0}{s}$ para indicar que en las transiciones el primer bit es p1 y el segundo p0 y que en los estados el valor de la salida corresponde a s. Esto último no sería estrictamente necesario en este circuito con una sola salida, pero es indispensable cuando existan más salidas.

El circuito arranca en el estado Reposo, donde espera a que se pulse la tecla p1. Cuando se pulse se irá al estado PulP1, donde se esperará su liberación, pasando al estado LibP1. En el caso hipotético que justo cuando se libera p1 se pulse p0 se iría directamente al estado PulP0, ya que estaríamos dentro de la secuencia p1, p0. Si antes de liberar p1 se pulsa p0 (11), esto se considera un error, por lo que se vuelve

[1] Otra alternativa que se mostrará más adelante es usar un detector de flanco para que cada pulsación de tecla genere un pulso de sólo un ciclo de reloj.

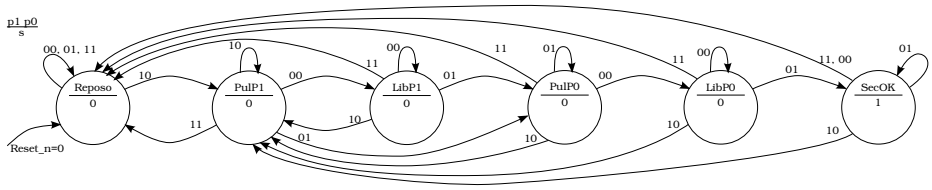


Figura 8.10: Diagrama de estados del detector de secuencia.

al estado de reposo. Desde el estado LibP1 se pasará a PulP0 cuando se pulse p0. Al igual que antes, si se pulsan las dos teclas a la vez se volverá al estado de reposo. Si por el contrario se pulsa p1 se considera que está empezando una nueva secuencia, por lo que se vuelve al estado PulP1. En el estado PulP0 se esperará a la liberación de p0 para pasar a LibP0. Al igual que antes la pulsación de las dos teclas lleva el circuito al estado de Reposo y la pulsación de p1 simultánea a la liberación de p0 llevará al circuito a PulP1. El comportamiento del circuito en el estado LibP0 es similar al del estado LibP1, sólo que ahora se espera a que se pulse p0 para pasar al estado SecOK donde se activará la salida para abrir la cerradura. En este estado se permanecerá mientras siga pulsada p0. Cuando se libere o se pulsen las dos teclas volveremos a Reposo y si se libera p0 y a la vez se pulsa p1 (cosa que es muy difícil que ocurra en un solo ciclo de reloj) se irá al estado PulP1 por considerarse el principio de una nueva secuencia.

8.5.1. Descripción en VHDL

La descripción en VHDL es similar a la realizada para el detector de flanco. Al igual que antes la descripción se divide en tres **process**. El primero de ellos contiene la descripción de las variables de estado y la única diferencia con respecto al detector de flanco es el estado inicial, que ahora es Reposo en lugar de Esp1. El segundo **process** contiene la función de transición de estados, que se obtiene directamente a partir del diagrama de estados. Por último, el tercero **process** contiene la función de salida.

```
1  -- Detector de secuencia para una cerradura electrónica.
2  -- Detecta la secuencia p1, p0, p0.
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6
7  entity DetectorSecuencia is
8
9    port (
10     reset_n : in  std_logic;            -- Inicialización
11     clk     : in  std_logic;            -- Reloj del sistema
```

```
12      p1, p0  : in  std_logic;            -- Pulsadores
13      s       : out std_logic);           -- salida
14
15  end DetectorSecuencia;
16
17  architecture behavioral of DetectorSecuencia is
18    -- Tipo para los estados
19    type   t_estados is (Reposo, PulP1, LibP1, PulP0,
20                         LibP0, SecOK);
21    -- Estados actual y siguiente
22    signal estado_act, estado_sig : t_estados;
23  begin  -- behavioral
24
25    VarEstado : process (clk, reset_n, estado_sig)
26    begin
27      if reset_n = '0' then               -- Reset asíncrono
28        estado_act <= Reposo;             -- (activo bajo)
29      elsif clk'event and clk = '1' then  -- Flanco de subida
30        estado_act <= estado_sig;
31      end if;
32    end process VarEstado;
33
34    TransicionEstados : process (estado_act, p1, p0)
35    begin
36      -- Por defecto nos quedamos en el estado actual
37      estado_sig <= estado_act;
38      case estado_act is
39        when Reposo =>
40          if p1 = '1' and p0 = '0' then
41            estado_sig <= PulP1;
42          end if;
43        when PulP1 =>
44           if p1 = '0' and p0 = '0' then
45             estado_sig <= LibP1;
46           elsif p1 = '0' and p0 = '1' then
47             estado_sig <= PulP0;
48           elsif p1 = '1' and p0 = '1' then
49             estado_sig <= Reposo;
50           end if;
51        when LibP1 =>
52          if p1 = '0' and p0 = '1' then
53            estado_sig <= PulP0;
54          elsif p1 = '1' and p0 = '0' then
55            estado_sig <= PulP1;
```

```
56          elsif p1 = '1' and p0 = '1' then
57            estado_sig <= Reposo;
58          end if;
59        when PulP0 =>
60          if p1 = '0' and p0 = '0' then
61            estado_sig <= LibP0;
62          elsif p1 = '1' and p0 = '0' then
63            estado_sig <= PulP1;
64          elsif p1 = '1' and p0 = '1' then
65            estado_sig <= Reposo;
66          end if;
67        when LibP0 =>
68          if p1 = '0' and p0 = '1' then
69            estado_sig <= SecOK;
70          elsif p1 = '1' and p0 = '0' then
71            estado_sig <= PulP1;
72          elsif p1 = '1' and p0 = '1' then
73            estado_sig <= Reposo;
74          end if;
75        when SecOK =>
76          if p1 = '0' and p0 = '1' then
77            estado_sig <= SecOK;
78          elsif p1 = '1' and p0 = '0' then
79            estado_sig <= PulP1;
80          else
81            estado_sig <= Reposo;
82          end if;
83        when others =>
84          estado_sig <= Reposo;
85      end case;
86    end process;
87
88    Salidas : process (estado_act)
89    begin
90      -- Por defecto la salida vale cero
91      s <= '0';
92      case estado_act is
93        when Reposo =>
94          null;
95        when PulP1 =>
96          null;
97        when LibP1 =>
98          null;
99        when PulP0 =>
```

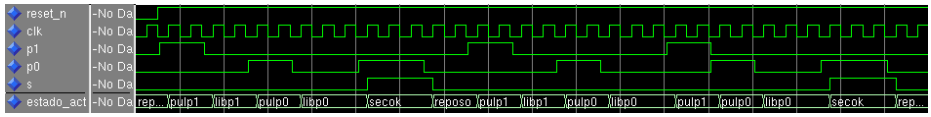


Figura 8.11: Simulación del detector de secuencia.

```
100         null;
101       when LibP0 =>
102         null;
103       when SecOK =>
104         s <= '1';
105       when others =>
106         null;
107     end case;
108   end process;
109
110 end behavioral;
```

El resultado de la simulación se muestra en la figura 8.11. En la simulación se ha introducido en primer lugar una secuencia correcta. Observe que justo cuando se pulsa `p0` por segunda vez la salida `s` se activa, manteniéndose activa mientras dura la pulsación de `p0`. Seguidamente se introduce una secuencia incorrecta (`p1`, `p0` y `p1`), quedándose la salida sin activar. No obstante este último `p1` es el inicio de una secuencia correcta, por lo que después de dos pulsaciones de `p0` la salida se vuelve a activar.

8.6. Detector de secuencia usando detectores de flanco

El detector de secuencia anterior necesita dos estados por cada pulsación de tecla que se desee detectar. El problema de esta solución es que si la secuencia es más larga se necesitan muchos más estados (dos por cada pulsación). Así, si se desea detectar la secuencia P1, P0, P1, P1 y P0 serían necesarios 10 estados. Para solucionarlo imaginemos que podemos pulsar una tecla durante sólo un ciclo de reloj. En ese caso sólo se necesita un estado por cada tecla de la secuencia, ya que cuando se detecte una tecla se realizará la transición de estado y en el ciclo de reloj siguiente, como la tecla ya estará liberada, esperaremos la pulsación de la siguiente tecla. Como es imposible que un usuario pulse una tecla durante sólo un ciclo de reloj, el efecto anterior se consigue poniendo un detector de flanco entre las entradas del circuito y la máquina de estados, tal como se muestra en la figura 8.12.

Según esto, el diagrama de estados para la secuencia P1, P0, P0 sería el mostrado en la figura 8.13. En primer lugar destacar que en el caso hipotético de que se pulsen las dos teclas a la vez lo consideraremos un error e iremos desde cualquier estado en el que se produzca dicha pulsación simultánea al estado Reposo. Por otro lado, el

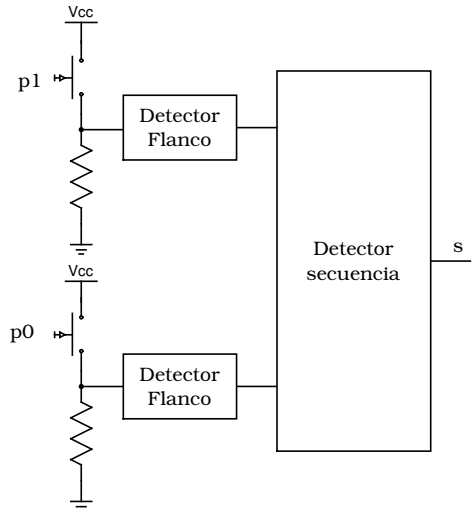


Figura 8.12: Diagrama de bloques del detector de secuencia usando detectores de flanco.

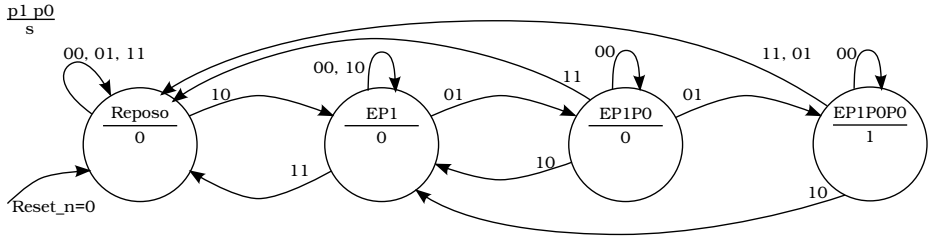


Figura 8.13: Diagrama de estados del detector de secuencia usando detectores de flanco.

paso del estado Reposo al EP1 se da cuando se pulse la tecla p1. Mientras no se pulse otra tecla seguiremos en este estado. También si se vuelve a pulsar p1 seguiremos en este estado, pues seguimos teniendo la primera tecla de la secuencia. Si se pulsa p0 pasaremos al estado EP1P0, esperando ahí mientras no se pulse otra tecla. Si se vuelve a pulsar p0 pasaremos al estado EP1P0P0 donde se activará la salida pues se ha completado la secuencia. Si desde cualquiera de estos estados se pulsa p1 volveremos al estado EP1, pues esta pulsación puede ser el comienzo de una nueva secuencia.

8.6.1. Descripción en VHDL

A continuación se muestra la descripción del circuito en VHDL. En primer lugar destacar que en la línea 26 se ha creado una señal denominada `entradas` para almacenar la salida de los detectores de flanco y facilitar así las comparaciones en el **process** que calcula las transiciones de estados. Estas señales se han conectado en las líneas 43 y 50, donde se han instanciado los detectores de flanco. El detector instanciado es el mostrado en la sección 8.4.4. El resto del código es muy parecido al del circuito anterior. Tan solo cambian las comparaciones para realizar las transiciones al haber agrupado todas las entradas en un vector. También se ha aprovechado este ejemplo para mostrar otra forma de describir la función de salida. En lugar de usar un **process** se ha usado una sentencia **when-else** para asignar a la salida un 1 cuando el estado actual es EP1P0P0.

```
1  -- Detector de secuencia para una cerradura electrónica.
2  -- Detecta la secuencia p1, p0, p0 usando detectores de
3  -- flanco en las entradas para simplificar la máquina de
4  -- estados.
5
6  library ieee;
7  use ieee.std_logic_1164.all;
8
9  entity DetectorSecuenciaFl is
10
11   port (
12     reset_n : in  std_logic;            -- Inicialización
13     clk     : in  std_logic;            -- Reloj del sistema
14     p1, p0  : in  std_logic;            -- Pulsadores
15     s       : out std_logic);           -- salida
16
17 end DetectorSecuenciaFl;
18
19 architecture behavioral of DetectorSecuenciaFl is
20   -- Tipo para los estados
21   type   t_estados is (Reposo, EP1, EP1P0, EP1P0P0);
22   -- Estados actual y siguiente
23   signal estado_act, estado_sig : t_estados;
24   -- Se crea esta señal para agrupar las dos entradas en
25   -- un vector
26   signal entradas : std_logic_vector(1 downto 0);
27
28   component DetectorFlanco
29     port (
30       e       : in  std_logic;
31       reset_n : in  std_logic;
```

```
32      clk     : in  std_logic;
33      s       : out std_logic);
34  end component;
35 begin  -- behavioral
36
37  -- instanciamos los detectores de flanco.
38  DetectorFlanco_1: DetectorFlanco
39      port map (
40          e       => p1,
41          reset_n => reset_n,
42          clk     => clk,
43          s       => entradas(1) );
44
45  DetectorFlanco_2: DetectorFlanco
46      port map (
47          e       => p0,
48          reset_n => reset_n,
49          clk     => clk,
50          s       => entradas(0) );
51
52  VarEstado : process (clk, reset_n, estado_sig)
53  begin
54    if reset_n = '0' then               -- Reset asíncrono
55      estado_act <= Reposo;             -- (activo bajo)
56    elsif clk'event and clk = '1' then  -- Flanco de subida
57      estado_act <= estado_sig;
58    end if;
59  end process VarEstado;
60
61  TransicionEstados : process (estado_act, entradas)
62  begin
63    -- Por defecto nos quedamos en el estado actual
64    estado_sig <= estado_act;
65    case estado_act is
66      when Reposo =>
67        if entradas = "10" then
68          estado_sig <= EP1;
69        end if;
70      when EP1 =>
71         if entradas = "01" then
72           estado_sig <= EP1P0;
73         elsif entradas = "11" then
74           estado_sig <= Reposo;
75         end if;
```

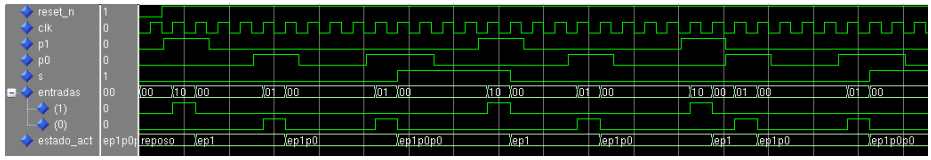
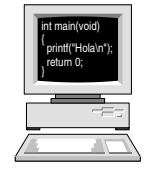


Figura 8.14: Simulación del detector de secuencia usando detectores de flanco.

```
76        when EP1P0 =>
77          if entradas = "01" then
78            estado_sig <= EP1P0P0;
79          elsif entradas = "10" then
80            estado_sig <= EP1;
81          elsif entradas = "11" then
82            estado_sig <= Reposo;
83          end if;
84        when EP1P0P0 =>
85          if entradas = "10" then
86            estado_sig <= EP1;
87          elsif entradas = "01" or entradas = "11" then
88            estado_sig <= Reposo;
89          end if;
90        when others =>
91          estado_sig <= Reposo;
92      end case;
93    end process;
94
95    -- Descripción de la función de salida.
96    s <= '1' when estado_act = EP1P0P0 else
97         '0';
98
99  end behavioral;
```

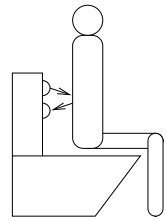
Una simulación del circuito se muestra en la figura 8.14. En la simulación se observa cómo el detector de flanco genera un pulso de un ciclo de reloj que hace que la máquina de estados realice la transición correspondiente. Por lo demás la simulación es idéntica a la mostrada en la sección anterior. En primer lugar se introduce una secuencia correcta, activándose la salida. La principal diferencia en este caso es que la salida permanece activa hasta que se pulsa una nueva tecla.



Realice el ejercicio 3

8.7. Ejercicios

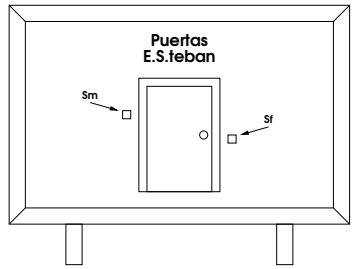
1. Diseñe un circuito que genere un pulso de salida cuando se detecte un flanco de bajada en su señal de entrada. Ha de usar una máquina de estados de Moore. El diseño ha de incluir un diagrama de estados, las tablas de transición, excitación y de salidas y el circuito final. Ha de usar *flip-flops* tipo D.
2. Diseñe un circuito que genere un pulso de salida cuando se detecte un flanco de bajada en su señal de entrada. Ha de usar una máquina de estados de Mealy. El diseño ha de incluir un diagrama de estados, la tabla de excitación y de salidas y el circuito final. Ha de usar *flip-flops* tipo J-K.
3. Modifique el detector de secuencia usando detectores de flanco expuesto en la sección 8.6 para que la salida del circuito esté activa sólo durante un ciclo de reloj.
4. La empresa Roca le ha encargado el diseño del circuito de control de una cisterna automática. El circuito detectará cuando alguien se acerque al WC a hacer sus necesidades y descargará la cisterna cuando éste haya terminado y se vaya del servicio. Para ello el WC contará con un emisor y un receptor de infrarrojos, tal como se muestra en la figura.

   El emisor tendrá una entrada (`Act_Emisor`) para activarlo y una salida (`Rec`) que se pondrá a 1 lógico cuando reciba la luz infrarroja reflejada en el usuario o a 0 cuando no reciba nada. La cisterna dispondrá de una electroválvula que se controlará con la señal `Cisterna`, de forma que cuando dicha señal esté a 1 se abrirá la electroválvula para descargar la cisterna.

   El funcionamiento del circuito será el siguiente: en el estado inicial, para evitar consumir demasiada energía, el emisor de infrarrojos estará desactivado. De este estado se irá a otro en el que se activará el emisor. Si no se detecta a nadie se volverá al estado inicial para volver a repetir el ciclo. Si se detecta a alguien, se esperará a que éste se marche para descargar la cisterna y volver al estado inicial.

   En el diseño ha de incluir el diagrama de estados y la descripción en VHDL del circuito.
5. El profesor Bacterio le ha pedido ayuda para que le diseñe el circuito de control de la nueva puerta secreta que se va a instalar en el cuartel general de la T.I.A. La puerta secreta ha sido convenientemente disimulada en un cartel publicitario de un conocido fabricante de puertas, tal como se muestra en la figura siguiente:



La puerta secreta está diseñada para ser usada única y exclusivamente por la pareja de superagentes más famosa de la agencia: Mortadelo y Filemón. Para ello se han colocado dos sensores a ambos lados de la puerta justo a la altura de la cabeza de cada uno de los dos agentes, representados en la figura mediante `Sm` y `Sf`. El funcionamiento de la puerta secreta es el siguiente: para activar el mecanismo Mortadelo y Filemón han de colocarse a ambos lados de la puerta, justo delante de su sensor correspondiente. Cuando el sistema detecte esta situación se encenderán dos luces que se han disimulado justo detrás de las letras "`E.`" y "`S.`" del cartel. A continuación Filemón tendrá que colocarse delante de la puerta para que el sensor `Sf` deje de detectarlo. Para confirmar este paso el circuito apagará la letra "S". Cuando Mortadelo vea que se ha apagado la luz se pondrá también delante de la puerta para que el sensor `Sm` deje de detectarlo. El circuito entonces activará una señal para abrir la puerta y otra para avisar al Súper de la llegada de sus agentes favoritos, de forma que pueda encomendarles una nueva y peligrosa misión. Ambas señales sólo deben durar un ciclo de reloj. Una vez iniciada la secuencia, si en algún paso se realiza una acción equivocada, como por ejemplo que sea Mortadelo en lugar de Filemón el primero en ponerse delante de la puerta, se activará una alarma para alertar a todos los agentes de la T.I.A. de un posible intento de asalto a su sede. Dicha alarma seguirá activa hasta que se vuelva a inicializar el circuito con la señal de reset.

Notas:

- Los sensores `Sm` y `Sf` dan un 1 cuando detectan una persona enfrente de ellos y un 0 en caso contrario.
- El diseño ha de incluir el diagrama de estados y la descripción en VHDL del circuito.

CAPÍTULO 9

Registros

9.1. Introducción

Un registro no es más que un conjunto de biestables que almacenan una palabra. Los más usados son los de entrada y salida en paralelo, que almacenan una palabra de n bits. No obstante en sistemas de comunicaciones son muy útiles los denominados registros de desplazamiento, que como veremos nos permiten enviar una palabra de n bits por un solo cable, enviando los bits en serie de uno en uno.

9.2. Registros de entrada y salida en paralelo

Estos registros están compuestos por n biestables de tipo D y almacenan una palabra de n bits (uno por biestable obviamente). En la figura 9.1 se muestra un registro de 4 bits. Como se puede observar el registro no es más que 4 biestables con las entradas de reloj y clear unidas. Por tanto, en cada flanco de reloj la entrada de datos se copiará a la salida, donde permanecerá estable hasta el siguiente flanco de reloj.

Descripción en VHDL

La descripción en VHDL de un registro se realiza en un **`process`**, de una forma similar a como se han descrito los registros que almacenan el estado en las máquinas

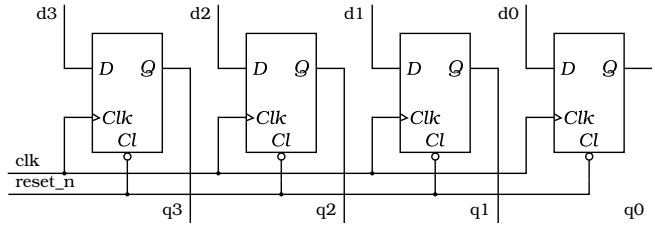


Figura 9.1: Registro con entrada y salida en paralelo.

de estado del capítulo anterior.

```
1  -- Registro de 4 bits
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity Registro is
7
8    port (
9      clk     : in  std_logic;         -- Entradas de control
10     reset_n : in  std_logic;
11     d       : in  std_logic_vector(3 downto 0);  --  datos
12     q       : out std_logic_vector(3 downto 0));
13
14 end Registro;
15
16 architecture behavioral of Registro is
17
18 begin  -- behavioral
19
20   process(clk, reset_n, d)
21   begin
22     if reset_n = '0' then
23       q <= (others => '0');
24     elsif clk'event and clk = '1' then
25       q <= d;
26     end if;
27   end process;
28
29 end behavioral;
```

9.2.1. Registros con control de carga

El registro anterior actualiza su salida en cada flanco de reloj. No obstante existen numerosas aplicaciones en las que es necesario controlar mediante una señal de *enable* cuando ha de almacenarse un nuevo valor en el registro. En este caso el circuito resultante es el mostrado en la figura 9.2. Como puede observar se ha añadido un multiplexor que controla si cuando llegue el flanco de reloj ha de cargarse en el registro el dato de entrada o el valor que está almacenado en el registro.

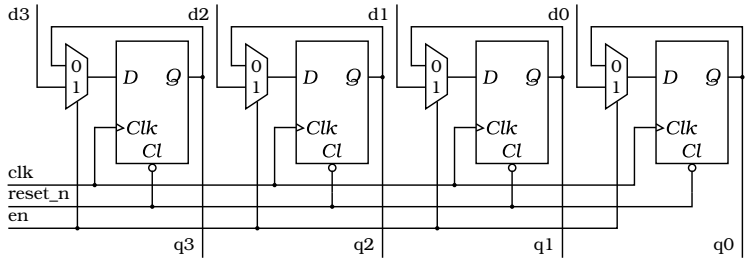


Figura 9.2: Registro con entrada y salida en paralelo y control de carga.

Descripción en VHDL

En este caso se añade a la descripción anterior un **if** (líneas 26–28) para que la salida sólo se actualice cuando la señal de *enable* (en) esté activa:

```
1  -- Registro de 4 bits con óhabilitacin de la carga
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5
6  entity RegistroEn is
7
8    port (
9      clk     : in  std_logic;         -- Entradas de control
10     reset_n : in  std_logic;
11     en      : in  std_logic;          -- óHabilitacin carga
12     d       : in  std_logic_vector(3 downto 0);  --  datos
13     q       : out std_logic_vector(3 downto 0));
14
15 end RegistroEn;
16
17 architecture behavioral of RegistroEn is
18
19 begin  -- behavioral
20
21   process(clk, reset_n, en, d)
22   begin
23     if reset_n = '0' then
24       q <= (others => '0');
25     elsif clk'event and clk = '1' then
26       if en = '1' then
27         q <= d;
28       end if;
```

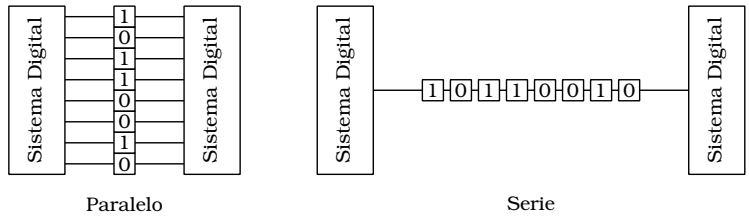


Figura 9.3: Transmisión paralelo y transmisión serie de datos.

```
29      end if;
30    end process;
31
32  end behavioral;
```

9.3. Registros de desplazamiento

En un sistema digital se trabaja normalmente con palabras de n bits, sobre las que se hacen una serie de operaciones. Si se desea transmitir una de estas palabras a otro sistema una solución es emplear n cables y por cada uno de ellos enviar un bit. Por ejemplo las primeras impresoras se conectaban al ordenador mediante un cable paralelo de 8 bits, por el que se podía enviar un byte de una vez. La otra alternativa es conectar los dos sistemas mediante un cable y por él enviar los n bits uno detrás de otro. Ambos métodos se ilustran gráficamente en la figura 9.3.

La conversión de datos de paralelo a serie y de serie a paralelo se realiza mediante los denominados registros de desplazamiento, los cuales se estudian en esta sección.

9.3.1. Registro de entrada serie y salida en paralelo

Este registro permite convertir una secuencia de n bits que entran en serie en una palabra de n bits. El diagrama del circuito (para palabras de 4 bits) se muestra en la figura 9.4. Como puede observar, consta de 4 *flip-flops* en los que la salida de cada uno se conecta a la entrada del siguiente, de forma que en cada flanco de reloj la palabra binaria que almacena se desplaza a la derecha. El hueco dejado por la palabra en el bit 3 se rellena con el valor de la entrada serie `e_s` y el bit 0 desaparece. En la figura 9.5 puede ver un diagrama de tiempos de la simulación del funcionamiento del circuito.

Descripción en VHDL

Para describir el registro de desplazamiento en VHDL lo primero que necesitamos es crear una señal interna para almacenar el registro (a la que se ha denominado precisamente `registro`), ya que el lenguaje impide obtener el valor de una señal de

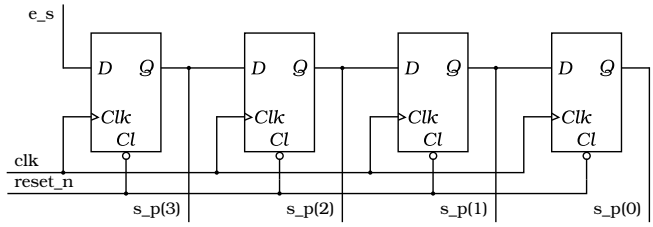


Figura 9.4: Registro serie–paralelo.

salida y como podemos ver en el esquema del circuito las salidas de los *flip-flops* también se conectan a las entradas de los siguientes.

La descripción de registro propiamente dicha es similar a la del registro paralelo–paralelo: cuando se activa el reset se pone el registro a cero y si el reset no está activo cuando se produce un flanco de subida del reloj se desplaza el contenido del registro. La descripción del desplazamiento se realiza en las líneas 26 y 27. En la primera se dice que la entrada serie se guarda en el bit tres del registro y en la segunda se dice que los bits 3 al 1 se guardan en los bits 2 al 0. Esta asignación se hace en el mismo orden en el que se describe, es decir, el bit 3 se conecta al 2, el bit 2 al 1 y el 1 al 0.

Por último, en la línea 31 se conecta el registro a la salida. Nótese que esta conexión se realiza fuera del **process**, aunque se podría haber realizado dentro obteniendo el mismo resultado. Se ha realizado así para dejar dentro del **process** solamente la descripción del registro.

```
1  -- Registro de desplazamiento con entrada serie y salida
2  -- paralelo de 4 bits
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6
7  entity RegSerPar is
8
9    port (
10     clk     : in  std_logic;
11     reset_n : in  std_logic;
12     e_s     : in  std_logic;                -- Entrada serie
13     s_p     : out std_logic_vector(3 downto 0));  -- salida
14
15 end RegSerPar;
16
17 architecture behavioral of RegSerPar is
18   signal registro : std_logic_vector(3 downto 0);
19 begin  -- behavioral
20
```

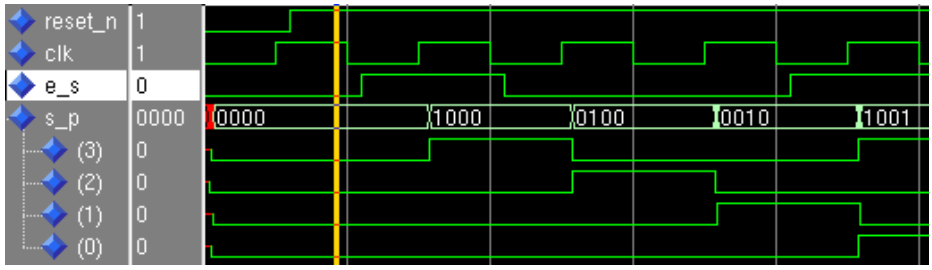
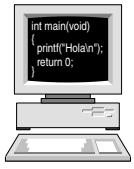


Figura 9.5: Simulación del registro con entrada serie y salida en paralelo.

```
21   Reg: process (clk, reset_n, e_s, registro)
22   begin  -- process Reg
23     if reset_n = '0' then
24       registro <= (others => '0');
25     elsif clk'event and clk = '1' then
26       registro(3) <= e_s;
27       registro(2 downto 0) <= registro(3 downto 1);
28     end if;
29   end process Reg;
30   -- Se copia el registro a la salida paralelo.
31   s_p <= registro;
32 end behavioral;
```



Realice el ejercicio 1

Una simulación del circuito descrito se encuentra en la figura 9.5.

9.3.2. Registro de entrada paralelo y salida en serie

Este registro realiza la función contraria al anterior: se carga una palabra en paralelo y se desplaza a la derecha en cada flanco de reloj. El diagrama del circuito para un registro de 4 bits es el mostrado en la figura 9.6.

En este registro existe una señal adicional denominada `c_d` que controla los multiplexores a la entrada de los *flip-flops*. Así, cuando esta señal vale 0 se conectan las entradas paralelo (`e_p`) a las entradas de los *flip-flops*, por lo que cuando llegue el flanco de subida del reloj se cargarán dichas entradas `e_p` en el registro. Cuando la señal `c_d` vale 1 se conecta a la entrada de cada *flip-flop* la salida del siguiente, excepto la entrada del *flip-flop* 3 que se conecta a tierra y la salida del *flip-flop* 0 que es la salida serie. De esta forma, en cada flanco de subida del reloj se desplazará el contenido del registro a la derecha, introduciendo ceros por el bit más significativo.

Una simulación del circuito se muestra en la figura 9.7.

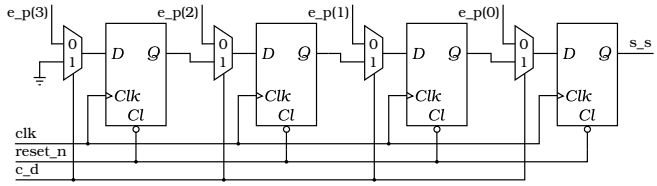


Figura 9.6: Registro paralelo–serie.

Descripción en VHDL

La descripción es similar a la del registro serie–paralelo. La única diferencia radica ahora en la descripción de la carga paralelo que se realiza mediante el **if** de la línea 27, de forma que cuando la señal c_d vale 0 se carga la entrada paralelo e_p en el registro y cuando vale 1 desplaza el contenido del registro, introduciendo un 0 por el bit 3.

```
1  -- Registro de desplazamiento con entrada paralelo de 4 bits
2  -- y salida serie
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6
7  entity RegParSer is
8
9    port (
10     clk     : in  std_logic;
11     reset_n : in  std_logic;
12     c_d     : in  std_logic;   -- Carga (0) o desplaza (1)
13     e_p     : in  std_logic_vector(3 downto 0); -- Entrada
14     s_s     : out std_logic);              -- Salida serie
15
16 end RegParSer;
17
18 architecture behavioral of RegParSer is
19   signal registro : std_logic_vector(3 downto 0);
20 begin  -- behavioral
21
22   Reg: process (clk, reset_n, registro, e_p)
23   begin
24     if reset_n = '0' then
25       registro <= (others => '0');
26     elsif clk'event and clk = '1' then
27       if c_d = '0' then                 -- Carga
```

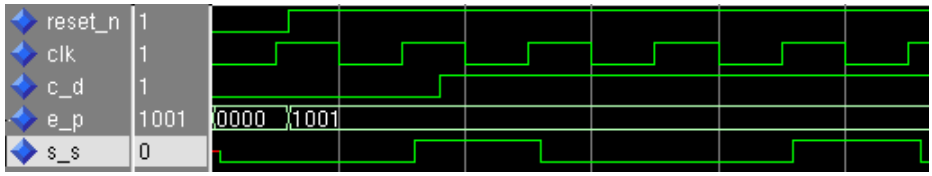
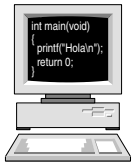


Figura 9.7: Simulación del registro con entrada paralelo y salida en serie.

```
28          registro <= e_p;
29        else                                  -- Desplaza
30          registro(3) <= '0'; -- Se introducen ceros.
31          registro(2 downto 0) <= registro(3 downto 1);
32        end if;
33      end if;
34    end process Reg;
35
36    -- Se copia el bit menos significativo a la salida serie.
37    s_s <= registro(0);
38  end behavioral;
```



Realice los ejercicios 2 y 3

9.3.3. *Registro de desplazamiento universal*

Este registro es una combinación de los tres anteriores, en el que además se añade la posibilidad de desplazar también a la izquierda. Para ello se usa ahora una señal de control `c` de dos bits que permite que el circuito tenga cuatro funcionalidades: desplazamiento a izquierda cuando la señal de control sea 00, mantenimiento cuando sea 01, carga paralelo cuando sea 10 y desplazamiento a derecha cuando sea 11.

El circuito se muestra en la figura 9.8. Para seleccionar el modo de funcionamiento del registro se usa un multiplexor en el que la entrada 0 se conecta a la salida del *flip-flop* de la derecha para realizar el desplazamiento a la izquierda, la 1 a la salida del propio *flip-flop* para mantener el valor, la 2 a la entrada paralelo para almacenar un nuevo valor en el registro y por último la 3 al *flip-flop* de la izquierda para realizar un desplazamiento a la derecha.

En el caso de los desplazamientos a izquierda se introduce un nuevo bit en el registro mediante la entrada serie que está a la derecha (`es_d`) y la salida serie `ss_i` coincide con el bit más significativo de la salida paralelo (`s_p(3)`). Del mismo modo cuando se desplaza a derecha se introduce por la izquierda el bit presente en la entrada `es_i` y la salida serie se realiza por `ss_d` que coincide con el bit menos significativo del registro.

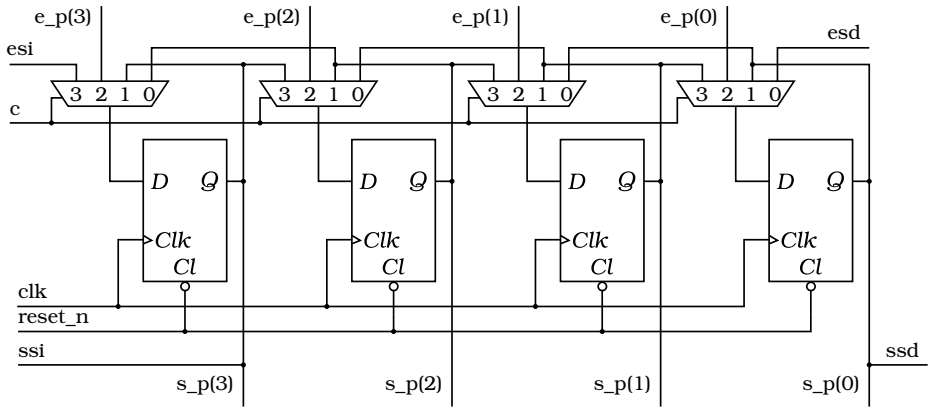


Figura 9.8: Registro de desplazamiento universal.

Descripción en VHDL

La descripción en VHDL es similar a la del resto de registros, sólo que ahora mediante el **if** de la línea 34 se selecciona qué ha de hacer el circuito en el flanco en función de la señal de control. Así cuando es 00 se desplaza a izquierda, cuando es 10 se realiza la carga paralelo y cuando es 11 se desplaza a derecha. Nótese que no se ha incluido en el **if** el valor 01 ya que en este caso el registro mantiene su valor y por tanto no hay que hacer nada.

```
1  -- Registro de desplazamiento con entrada paralelo de 4 bits
2  -- y posibilidad de desplazar a izquierda y derecha.
3  -- Mediante la entrada c se controla el funcionamiento del
4  -- circuito: c=00 Desplaza a Izq., c=01 Mantenimiento,
5  -- c=10 Carga, C=11 Desplaza a Der.
6
7  library ieee;
8  use ieee.std_logic_1164.all;
9
10 entity RegDespl is
11
12   port (
13     clk     : in  std_logic;
14     reset_n : in  std_logic;
15     c       : in  std_logic_vector(1 downto 0);  -- Control
16     esd     : in  std_logic;            -- entrada serie Der.
17     esi     : in  std_logic;            -- entrada serie Izq.
18     e_p     : in  std_logic_vector(3 downto 0);-- E. paralelo
19     s_p     : out std_logic_vector(3 downto 0);-- S. paralelo
```

```
20    ssd     : out std_logic;            -- salida serie Der.
21    ssi     : out std_logic);           -- salida serie Izq.
22
23 end RegDespl;
24
25 architecture behavioral of RegDespl is
26   signal registro : std_logic_vector(3 downto 0);
27 begin  -- behavioral
28
29   Reg: process (clk, reset_n, registro, e_p, c)
30   begin
31     if reset_n = '0' then
32       registro <= (others => '0');
33     elsif clk'event and clk = '1' then
34       if c = "00" then    -- Desplaza a Izq.
35         registro(0) <= esd;
36         registro(3 downto 1) <= registro (2 downto 0);
37       elsif c = "10" then -- Carga
38         registro <= e_p;
39       elsif c = "11" then -- Desplaza a Der.
40         registro(3) <= esi;
41         registro(2 downto 0) <= registro(3 downto 1);
42       end if;
43     end if;
44   end process Reg;
45
46   -- Se copia el bit menos significativo a la salida serie
47   -- Derecha.
48   ssd <= registro(0);
49   -- El más significativo a la salida serie Izquierda
50   ssi <= registro(3);
51   -- Y el registro a la salida paralelo
52   s_p <= registro;
53 end behavioral;
```

9.4. Ejercicios

1. Modifique el registro de entrada serie y salida en paralelo mostrado en la sección 9.3.1 para que el desplazamiento en lugar de realizarse en cada flanco de reloj se realice sólo cuando se indique mediante una señal de entrada, a la que se denominará `enable`. Así, cuando esta señal valga 0 el registro mantendrá su valor, incluso aunque lleguen flancos de reloj. Si por el contrario la señal vale 1 cuando llegue un flanco de reloj el circuito realizará un desplazamiento

de un bit a la derecha. Realice en primer lugar un esquema del circuito y a continuación descríbalo en VHDL.

2. Repita el ejercicio anterior para el registro de entrada en paralelo y salida en serie mostrado en la sección 9.3.2.

3. Modifique el código del ejercicio anterior para que el número de bits del registro se especifique mediante un genérico.

4. Se acaba de comprar un coche y para sorprender a sus amistades se le ocurre instalarle lo último en *tuning*: un juego de luces en la parte delantera al estilo del "coche fantástico". Para fardar todavía más, decide hacer usted mismo la electrónica de control. El circuito será capaz de controlar 6 lámparas. Al inicializarlo con la señal de reset se iluminará solo la lámpara izquierda. Después de medio segundo se apagará la lámpara izquierda y se encenderá la que le sigue. La secuencia completa será la mostrada a continuación:

   ```
   100000
   010000
   001000
   000100
   000010
   000001
   000010
   000100
   001000
   010000
   100000
   ```

   Para diseñar el circuito dispone de un reloj de 2 Hz. Realice en primer lugar un circuito usando puertas lógicas, *flip-flops*, multiplexores, etc. A continuación describa el circuito diseñado en VHDL. **Nota:** Es fácil implantar el circuito anterior mediante un registro de desplazamiento que pueda desplazar en los dos sentidos.

5. Después del éxito conseguido con el diseño del problema anterior, decide actualizar un poco el juego de luces para el *tuning* de su coche. El circuito será capaz de controlar 6 lámparas. Al inicializarlo con la señal de reset se apagarán todas las lámparas. Después de medio segundo se encenderá la lámpara izquierda, después de otro medio segundo se encenderá la que sigue, continuándose la secuencia hasta que se hayan encendido todas las luces. Entonces se continuará la secuencia apagando la luz derecha, luego la que está a su izquierda, etc. La secuencia completa será la mostrada a continuación:

```
000000
100000
110000
111000
111100
111110
111111
111110
111100
111000
110000
100000
000000
```

Para diseñar el circuito dispone de un reloj de 2 Hz. Realice en primer lugar un circuito usando puertas lógicas, *flip-flops*, multiplexores, etc. A continuación describa el circuito diseñado en VHDL. **Nota:** Es fácil implantar el circuito anterior mediante un registro de desplazamiento que pueda desplazar en los dos sentidos.

6. Diseñe un circuito digital para mostrar en dos *displays* la temperatura máxima y mínima. El valor de la temperatura lo proporciona un sensor en formato signo/magnitud de 7 bits. Además de la entrada de temperatura, existirá otra señal para inicializar el sistema poniendo la temperatura actual como máxima y mínima.

   Para diseñar el circuito suponga que dispone de comparadores para números de 7 bits en formato signo/magnitud.

   Realice en primer lugar un diagrama de bloques del circuito y a continuación describa el circuito en VHDL.

CAPÍTULO 10

Contadores

10.1. Introducción

Un contador no es más que un circuito secuencial que como su propio nombre indica es capaz de contar. No obstante existen varias formas de contar. La más natural es empezar en cero y continuar hasta un valor final, pero si estamos en una plataforma de lanzamiento de cohetes espaciales igual necesitamos un contador que cuente hacia atrás. Además hay contadores caprichosos, como los de los ascensores de los rascacielos americanos que del 12 pasan al 14 ya que el piso 13 no existe para no herir la sensibilidad de las personas supersticiosas.

En este capítulo vamos a estudiar todos estos tipos de contadores. Empezaremos por los contadores binarios ascendentes, los cuales cuentan según la secuencia $0, 1 \cdots (2^n - 1), 0, 1 \cdots$ para a continuación estudiar los descendentes, que obviamente realizan la secuencia $0, (2^n - 1), (2^n - 2) \cdots 1, 0, (2^n - 1) \cdots$. Veremos también cómo fusionar estos contadores en uno que pueda contar hacia arriba o hacia abajo según le indique una señal de control. A continuación estudiaremos cómo realizar un contador en BCD y su generalización a un contador módulo n, el cual cuenta de 0 hasta n y después reinicia la cuenta en 0. Por último estudiaremos los contadores de secuencia arbitraria por si algún día le contratan para diseñar el ascensor de un rascacielos.

10.2. Contador binario ascendente

Un contador binario ascendente de n bits es un circuito que realiza la cuenta $0, 1 \cdots (2^n - 1), 0, 1 \cdots$. Aunque su diseño puede realizarse mediante una máquina de estados, el circuito puede deducirse fácilmente a partir de su diagrama temporal. Así, si se fija en el diagrama de tiempos de la figura 10.1 verá que la salida `s(0)` cambia en cada flanco de reloj. Conseguir este comportamiento es fácil: basta con usar un *flip-flop* tipo T y conectar permanentemente su salida a 1, que es lo que se ha hecho en el circuito de la figura 10.1.

Si se fija ahora en la salida `s(1)` verá que ésta cambia sólo en los flancos de reloj

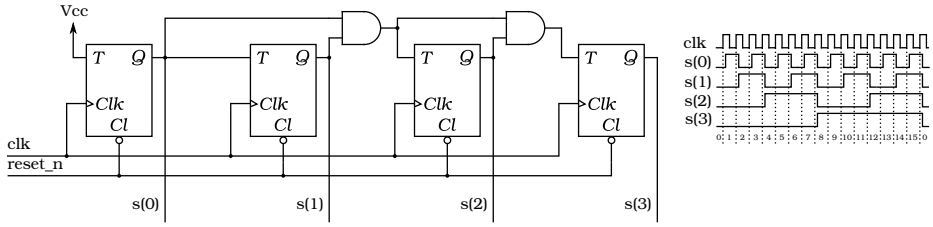


Figura 10.1: Contador binario ascendente.

en los que la salida `s(0)` vale 1.[1] Lo más fácil para generar la salida `s(1)` es usar un *flip-flop* tipo T y conectar su entrada a `s(0)`. La salida `s(2)` sólo cambia en los flancos en los tanto `s(0)` como `s(1)` valen 1. Por ello se ha generado mediante un *flip-flops* tipo T en el que se ha conectado a su entrada la AND de `s(0)` y `s(1)`. Por último, la salida `s(3)` cambia en los flancos en los que `s(0)`, `s(1)` y `s(2)` valen 1. Para generarla, al igual que antes, se ha usado un *flip-flops* tipo T en el que se ha conectado a su entrada la AND de `s(0)`, `s(1)` y `s(2)`. Para implementar esta AND se ha usado el resultado de la AND de `s(0)` y `s(1)`, al que se le ha hecho la AND con `s(2)`.

Si se fija en el circuito es fácil ampliarlo a un número mayor de bits. Basta con ir añadiendo un *flip-flop* tipo T y una puerta AND, conectándolos del mismo modo que se ha hecho para el último *flip-flop* del circuito de la figura 10.1.

10.2.1. Descripción en VHDL

La descripción del contador es mejor hacerla en alto nivel usando el operador + para incrementar la cuenta en 1 cada flanco de reloj. Si se fija en el código es similar al registro paralelo–paralelo estudiado en el capítulo anterior, sólo que ahora en cada flanco de reloj se incrementa el valor de la cuenta en 1.

También conviene destacar que al igual que los registros de desplazamiento, el valor del contador ha de almacenarse en una señal interna, ya que es necesario leer su valor. Esta señal se copia a la salida `s` al final de la arquitectura, de la misma forma que se hizo en el registro de desplazamiento.

```
1 -- Contador ascendente de 4 bits
2
3 library ieee;
4 use ieee.std_logic_1164.all;
5 use ieee.numeric_std.all;
6
7 entity ContadorAscendente is
```

[1] Fíjese que en este caso los retardos juegan a nuestro favor, pues justo cuando llega el flanco de reloj `s(0)` todavía vale 1 y será después del retardo de propagación del *flip-flop* cuando se ponga a cero.

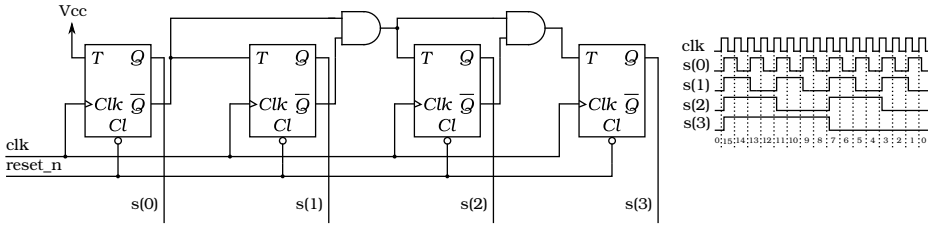
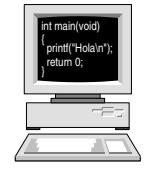


Figura 10.2: Contador binario descendente.

```
8
9   port (
10    clk     : in  std_logic;
11    reset_n : in  std_logic;
12    s       : out std_logic_vector(3 downto 0));  -- Salida
13
14 end ContadorAscendente;
15
16 architecture behavioral of ContadorAscendente is
17   signal contador : std_logic_vector(3 downto 0);
18 begin  -- behavioral
19
20   process (clk, reset_n)
21   begin  -- process
22     if reset_n = '0' then
23       contador <= (others => '0');
24     elsif clk'event and clk = '1' then
25       contador <= std_logic_vector(unsigned(contador) + 1);
26     end if;
27   end process;
28
29   s <= contador;
30 end behavioral;
```



Realice el ejercicio 1

10.3. Contador binario descendente

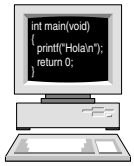
Como puede ver en el circuito de la figura 10.2, el contador descendente es muy similar al ascendente. Lo único que cambia es que las entradas T de los *flip-flops* están ahora controladas por las salidas $\overline{Q}$ de los *flip-flops* precedentes, en lugar de por las salidas Q. Ello es porque si se fija en el diagrama de tiempos del contador mostrado en la figura 10.2, la salida `s(1)` cambia sólo cuando `s(0)` es cero, la salida `s(2)` cambia sólo cuando `s(0)` y `s(1)` son cero y la salida `s(3)` sólo cambia cuando

`s(0)`, `s(1)` y `s(2)` son cero.

10.3.1. Descripción en VHDL

Como es de esperar la descripción en VHDL del contador descendente es casi igual a la del ascendente. Tan sólo se ha sustituido en la línea 25 el +1 por un -1 para que en cada ciclo de reloj se decremente el contador. También se ha modificado la línea 23 para que cuando se inicialice el contador lo haga a 1111 en lugar de a `0000`.

```
1  -- Contador descendente de 4 bits
2
3  library ieee;
4  use ieee.std_logic_1164.all;
5  use ieee.numeric_std.all;
6
7  entity ContadorDescendente is
8
9    port (
10     clk     : in  std_logic;
11     reset_n : in  std_logic;
12     s       : out std_logic_vector(3 downto 0));  -- Salida
13
14 end ContadorDescendente;
15
16 architecture behavioral of ContadorDescendente is
17   signal contador : std_logic_vector(3 downto 0);
18 begin  -- behavioral
19
20   process (clk, reset_n)
21   begin  -- process
22     if reset_n = '0' then
23       contador <= (others => '1');
24     elsif clk'event and clk = '1' then
25       contador <= std_logic_vector(unsigned(contador) - 1);
26     end if;
27   end process;
28
29   s <= contador;
30 end behavioral;
```



Realice el ejercicio 2

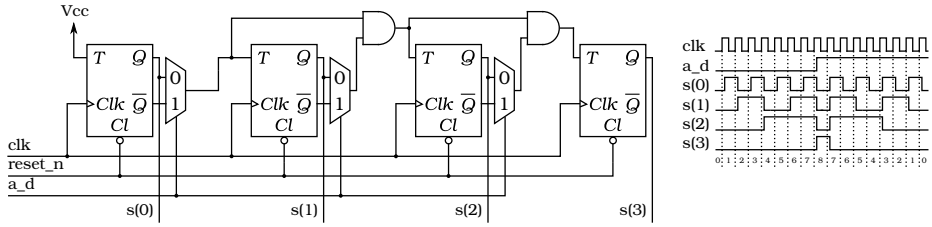


Figura 10.3: Contador binario ascendente/descendente.

10.4. Contador ascendente / descendente

En muchas situaciones son necesarios contadores que cuenten hacia arriba y hacia abajo. Por ejemplo, si queremos llevar la cuenta de los coches que hay en un aparcamiento público, cada vez que entra un coche es necesario incrementar el contador y cada vez que sale es necesario decrementarlo.

A partir de los dos contadores anteriores es fácil crear un contador ascendente/descendente. Lo único que hay que hacer para ello es conectar las salidas Q a las entradas T de los *flip-flops* cuando queramos un contador ascendente y conectar las salidas $\overline{Q}$ cuando queramos un contador descendente. Para controlar esta conexión se usa un multiplexor, el cual será controlado por una entrada al circuito. En la figura 10.3 se muestra el diagrama del contador. La señal de control del sentido de cuenta se ha denominado `a_d`. Como puede observar, cuando `a_d` valga 0 la cuenta será ascendente y cuando sea 1 la cuenta será descendente.

En la figura 10.3 también se muestra un diagrama de tiempos con un ejemplo de la evolución temporal del contador. En ella se ha supuesto que la entrada `a_d` es cero inicialmente, por lo que el contador cuenta hacia arriba. Cuando la cuenta llega a 8 se ha supuesto que la entrada `a_d` se pone a 1, por lo que el contador cuenta ahora hacia abajo.

10.4.1. Descripción en VHDL

Para describir en VHDL el contador ascendente/descendente se ha añadido al contador ascendente un **`if`** en la línea 26 para decidir en cada flanco de reloj si se incrementa la cuenta o se decrementa en función del valor de la entrada `a_d`.

```
1 -- Contador ascentente/descendente de 4 bits
2
3 library ieee;
4 use ieee.std_logic_1164.all;
5 use ieee.numeric_std.all;
6
7 entity ContadorAscDesc is
8
```

```
9   port (
10    clk     : in  std_logic;
11    reset_n : in  std_logic;
12    a_d     : in std_logic;             -- asc (0)/desc (1)
13    s       : out std_logic_vector(3 downto 0));  -- Salida
14
15 end ContadorAscDesc;
16
17 architecture behavioral of ContadorAscDesc is
18   signal contador : std_logic_vector(3 downto 0);
19 begin  -- behavioral
20
21   process (clk, reset_n)
22   begin  -- process
23     if reset_n = '0' then
24       contador <= (others => '0');
25     elsif clk'event and clk = '1' then
26       if a_d = '0' then
27         contador <= std_logic_vector(unsigned(contador) + 1);
28       else
29         contador <= std_logic_vector(unsigned(contador) - 1);
30       end if;
31     end if;
32   end process;
33
34   s <= contador;
35 end behavioral;
```

10.5. Contadores con habilitación de la cuenta

En muchas situaciones los contadores no tienen que contar en cada flanco de reloj. Si volvemos al ejemplo del contador de coches del aparcamiento, el contador sólo tendrá que cambiar su valor cuando entre o salga un coche. No obstante, recuerde que en un circuito síncrono todos los elementos han de estar conectados al mismo reloj, por lo que si se le ha ocurrido poner una puerta AND a la entrada del reloj para que sólo cuente cuando le interese ha elegido una pésima solución.

Para conseguir que el contador cuente sólo cuando nos interese, se añade una señal denominada `enable` de forma que cuando en un flanco de reloj esta señal esté a 1 el contador contará y cuando esté a 0 el contador mantendrá su valor.

Para describir este contador en VHDL basta con añadir la nueva señal a la declaración de puertos del componente (línea 14) y un nuevo **`if`** (linea 28) para que el contador sólo cambie su valor cuando la señal `enable` valga 1.

```
1  -- Contador ascendente/descendente de 4 bits con habilitación
2  -- de la cuenta
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  entity ContadorAscDescEnable is
9
10   port (
11     clk     : in  std_logic;
12     reset_n : in  std_logic;
13     a_d     : in  std_logic;           -- asc (0)/desc (1)
14     enable  : in  std_logic;           -- 1 cuenta, 0 mant
15     s       : out std_logic_vector(3 downto 0)); -- Salida
16
17 end ContadorAscDescEnable;
18
19 architecture behavioral of ContadorAscDescEnable is
20   signal contador : std_logic_vector(3 downto 0);
21 begin  -- behavioral
22
23   process (clk, reset_n)
24   begin  -- process
25     if reset_n = '0' then
26       contador <= (others => '0');
27     elsif clk'event and clk = '1' then
28       if enable = '1' then
29         if a_d = '0' then
30           contador <= std_logic_vector(unsigned(contador)+1);
31         else
32           contador <= std_logic_vector(unsigned(contador)-1);
33         end if;
34       end if;
35     end if;
36   end process;
37
38   s <= contador;
39 end behavioral;
```

Realice el ejercicio 3

10.6. Contadores módulo m

Los contadores expuestos hasta ahora, cuando la cuenta llegan a 2^n-1, siendo n el número de bits, rebosan y continúan la cuenta en cero. No obstante, en ocasiones es conveniente que la cuenta termine en un valor menor a 2^n-1. A este tipo de contadores se les denomina contadores módulo m, siendo m el número de valores por los que pasa el contador. Así un contador módulo 10 cuenta 0, $1 \cdots 9$, 0, $1 \cdots$ Si el contador es descendente la cuenta será obviamente al revés: 9, $8 \cdots 1$, 0, 9 $\cdots$.

El diseño de este tipo de contadores añade un comparador a la salida del contador que inicia la cuenta cuando se detecta que el contador ha llegado a su valor final. Por ejemplo, en el siguiente código se muestra un contador módulo 10 ascendente con habilitación de la cuenta.

```
1  -- Contador ascendente módulo 10 con habilitación
2  -- de la cuenta
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  entity ContadorAscModulo10 is
9
10   port (
11     clk     : in  std_logic;
12     reset_n : in  std_logic;
13     enable  : in  std_logic;           -- 1 cuenta, 0 mant
14     s       : out std_logic_vector(3 downto 0)); -- Salida
15
16 end ContadorAscModulo10;
17
18 architecture behavioral of ContadorAscModulo10 is
19   signal contador : std_logic_vector(3 downto 0);
20 begin  -- behavioral
21
22   process (clk, reset_n)
23   begin  -- process
24     if reset_n = '0' then
25       contador <= (others => '0');
26     elsif clk'event and clk = '1' then
27       if enable = '1' then
28         if contador = "1001" then
29           contador <= (others => '0');
30         else
31           contador <= std_logic_vector(unsigned(contador)+1);
```

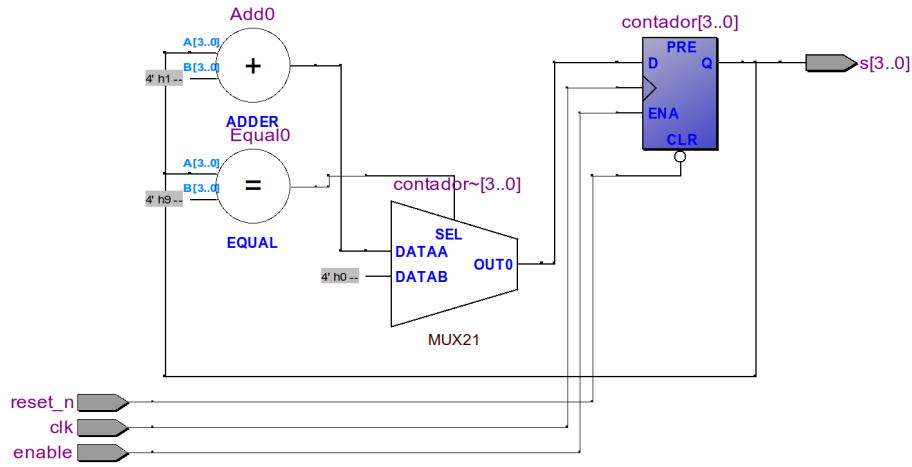


Figura 10.4: Contador ascendente módulo 10.

```
32          end if;
33        end if;
34      end if;
35    end process;
36
37    s <= contador;
38  end behavioral;
```

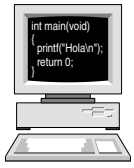
El circuito sintetizado a partir de la descripción anterior se muestra en la figura 10.4. A la vista de la figura conviene destacar varias cosas:

- En las FPGA sólo existen *flip-flops* tipo D, por lo que el contador se implanta como n *flip-flops* tipo D para almacenar los n bits de la cuenta y un sumador para sumar 1.
- El final de la cuenta se detecta con un comparador con 9, de forma que cuando la comparación es cierta, mediante un multiplexor se pone el valor 0 en la entrada D de los *flip-flops*. Así en el siguiente flanco el contador pasará a valer 0. Cuando la cuenta es distinta de 9 el multiplexor conecta a la entrada D la salida del sumador, que contiene el valor de la cuenta mas 1. De esta forma se incrementará el contador en el siguiente flanco de reloj.
- Los *flip-flops* tipo D tienen una entrada de habilitación `ENA`, la cual cuando vale 0 mantiene el valor almacenado en el *flip-flop* aunque llegue un flanco de reloj y cuando vale 1 permite la carga de la entrada D en el *flip-flop* cuando llega un flanco de reloj.

10.6.1. Contadores módulo m descendentes

En el caso de un contador descendente, la descripción es muy similar, salvo que ahora cuando el contador llega a cero es necesario cargar el valor del módulo menos 1. Así, si queremos describir un contador descendente módulo 10, basta con cambiar el código dentro del **process** a:

```
if reset_n = '0' then
     contador <= (others => '0');
   elsif clk'event and clk = '1' then
     if enable = '1' then
       if contador = "0000" then
         contador <= "1001";
       else
         contador <= std_logic_vector(unsigned(contador)-1);
       end if;
     end if;
   end if;
 end process;
```



Realice el ejercicio 4

10.7. Conexión de contadores en cascada

Si se diseñan contadores binarios en VHDL es fácil aumentar el número de bits del contador, sobre todo si se diseñan de forma genérica, tal como se ha propuesto en los ejercicios 1 y 2. No obstante, cuando se diseñan contadores usando lógica discreta o si se diseñan contadores en decimal es necesario poder conectarlos en cascada de forma que un contador cuente cuando el contador anterior rebose. Para ilustrar esto nada mejor que poner un ejemplo. Supongamos que queremos diseñar un contador ascendente que cuente en BCD natural. En la sección anterior hemos visto cómo diseñar un contador que cuente de 0 a 9, con lo que tenemos resuelto el cómo hacer un contador de una cifra. Si queremos hacer uno de tres cifras necesitamos tres contadores, uno para las unidades, otro para las decenas y otro para las centenas. El problema está en que el contador de las decenas sólo ha de incrementarse cuando rebose el contador de unidades, es decir, cuando éste pase de 9 a 0. De la misma forma, el contador de centenas sólo ha de incrementarse cuando rebose el de decenas.

En definitiva, para poder conectar contadores en cascada lo único que hay que añadir a los contadores expuestos hasta ahora es una salida de acarreo que genere un pulso de un ciclo de reloj cuando el contador rebose y conectar dicha salida al `enable` del contador de la cifra siguiente. Por ejemplo, en la figura 10.5 se muestra el esquema de un contador decimal de tres cifras construido a partir de tres contadores módulo 10.

El contador módulo 10 con acarreo de salida se describe en VHDL de forma similar al contador módulo 10 anterior:

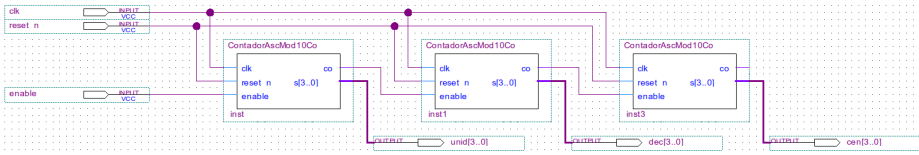


Figura 10.5: Contador ascendente decimal de tres cifras.

```vhdl
-- Contador ascendente módulo 10 con habilitación
-- de la cuenta y salida de acarreo.

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ContadorAscMod10Co is

  port (
    clk     : in  std_logic;
    reset_n : in  std_logic;
    enable  : in  std_logic;            -- 1 cuenta, 0 mant
    co      : out std_logic;            -- acarreo
    s       : out std_logic_vector(3 downto 0)); -- Salida

end ContadorAscMod10Co;

architecture behavioral of ContadorAscMod10Co is
  signal contador : std_logic_vector(3 downto 0);
begin  -- behavioral

  process (clk, reset_n)
  begin  -- process
    if reset_n = '0' then
      contador <= (others => '0');
    elsif clk'event and clk = '1' then
      if enable = '1' then
        if contador = "1001" then
          contador <= (others => '0');
        else
          contador <= std_logic_vector(unsigned(contador)+1);
        end if;
      end if;
    end if;
```

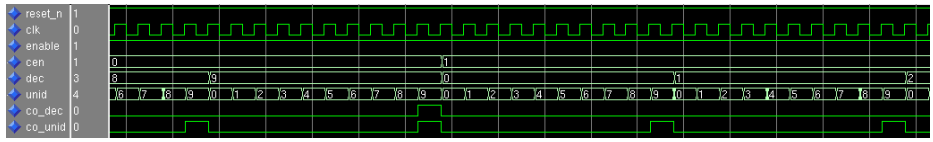


Figura 10.6: Simulación del contador ascendente decimal de tres cifras.

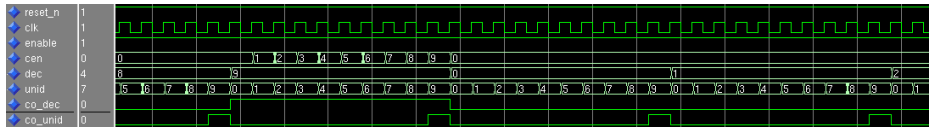


Figura 10.7: Simulación del contador ascendente decimal de tres cifras **erróneo**.

```
36   end process;
37
38   co <= '1' when contador = "1001" and enable = '1' else
39         '0';
40   s <= contador;
41 end behavioral;
```

Si compara ambas descripciones, los únicos cambios han sido el añadir la salida `co` a la declaración de puertos (línea 14) y su lógica en la línea 38. Nótese que la salida se pone a 1 sólo cuando el contador valga 9 y se active la señal de `enable`, ya que justo cuando se de esa condición, en el siguiente flanco de reloj se producirá el rebose. Así, al valer el acarreo de salida 1 en dicho flanco también se incrementará el contador siguiente. Para ilustrar este proceso en la figura 10.6 se muestra la simulación del paso del contador de 99 a 100. Nótese que si sólo hubiéramos puesto en la condición de la línea 38 el que el contador sea 9, la señal `co` estaría activa desde que el contador llega a 90 hasta que llega a 99, con lo que el contador de decenas se incrementaría en cada flanco. Para ilustrar este comportamiento anómalo en la figura 10.6 se muestra la misma simulación, pero realizada con un contador módulo 10 en el que la línea 38 se ha sustituido por:

```
co <= '1' when contador = "1001" else '0';
```

Nótese que en este caso, al estar la señal de `enable` del contador de centenas habilitada mientras las decenas valen 9, el contador de centenas cuenta 10 veces y vuelve a situarse en 0, con lo cual no conseguiremos contar ninguna centena.

10.7.1. Generación del acarreo en contadores descendentes

Si el contador es descendente, el acarreo a la siguiente cifra es necesario generarlo en el paso de 0 a 9, pues es en ese momento en el que la siguiente cifra ha

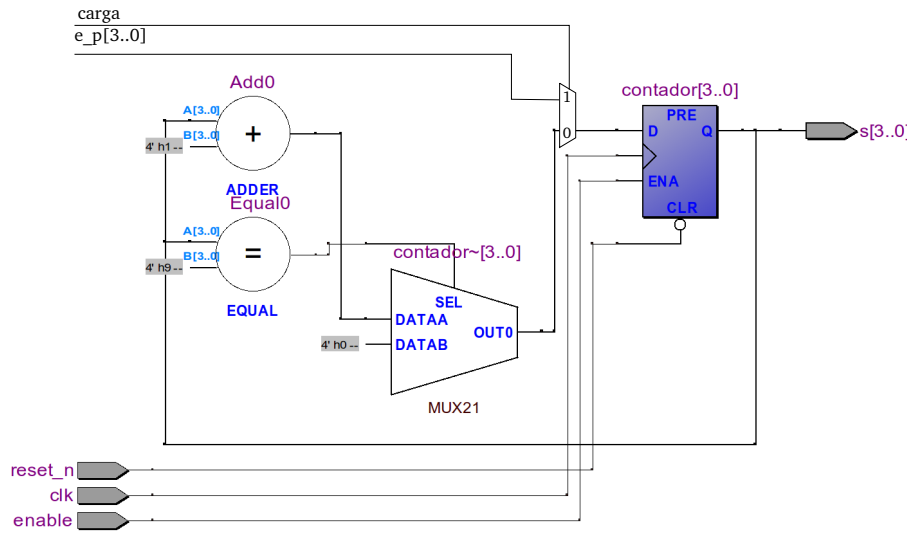
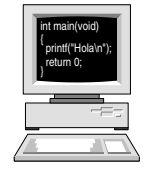


Figura 10.8: Contador ascendente módulo 10 con carga paralelo.

de decrementarse. Así, si el contador anterior fuese decreciente, la salida `co` se describiría de la siguiente manera:



Realice los ejercicios 5, 6 y 7

```
co <= '1' when contador = "0000" and enable = '1' else
      '0';
```

10.8. Contadores con carga paralelo

En los contadores mostrados hasta ahora existe una señal de reset que los inicializa a cero, pero a partir de este momento el contador sólo puede incrementarse o decrementarse. No obstante existen situaciones en las que es preciso poder cambiar el valor de la cuenta de un contador. Por ejemplo piense en un reloj digital. Si queremos ponerlo en hora es necesario poder modificar el valor de la cuenta.

Para la realización de la carga paralelo basta con añadir un multiplexor a la entrada de los *flip-flops* que almacenan el valor de la cuenta, tal como se muestra en la figura 10.8, en la que se ha añadido una carga paralelo al contador ascendente módulo 10 de la figura 10.4. Dicho multiplexor se controla mediante la señal `carga`, de forma que cuanto esta señal es cero el contador cuenta en cada flanco de reloj y cuando es 1 carga el valor presente en la entrada paralelo `e_p` en el contador cuando se produzca el flanco de reloj.

Descripción en VHDL

Para describir en VHDL del contador anterior lo único que hay que hacer es añadir en la descripción de los puertos las dos nuevas entradas (línea 14) y dentro del process un **if** para modelar el multiplexor (línea 30).

```
1  -- Contador ascendente módulo 10 con habilitación
2  -- de la cuenta, salida de acarreo y carga paralelo.
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  entity ContadorAscMod10Co is
9
10   port (
11     clk     : in  std_logic;
12     reset_n : in  std_logic;
13     enable  : in  std_logic;           -- 1 cuenta, 0 mant
14     carga   : in  std_logic;           -- carga paralelo
15     e_p     : in  std_logic_vector(3 downto 0); -- e. par.
16     co      : out std_logic;           -- acarreo
17     s       : out std_logic_vector(3 downto 0)); -- Salida
18
19 end ContadorAscMod10Co;
20
21 architecture behavioral of ContadorAscMod10Co is
22   signal contador : std_logic_vector(3 downto 0);
23 begin  -- behavioral
24
25   process (clk, reset_n)
26   begin  -- process
27     if reset_n = '0' then
28       contador <= (others => '0');
29     elsif clk'event and clk = '1' then
30       if carga = '1' then
31         contador <= e_p;
32       elsif enable = '1' then
33         if contador = "1001" then
34           contador <= (others => '0');
35         else
36           contador <= std_logic_vector(unsigned(contador)+1);
37         end if;
38       end if;
39     end if;
```

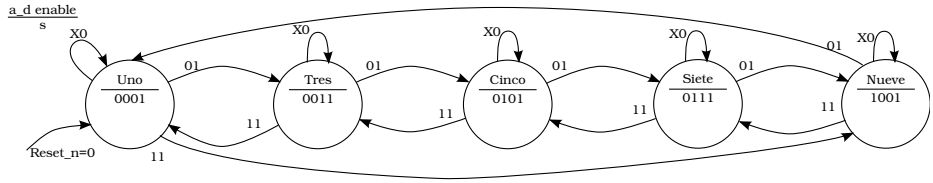
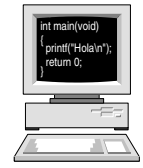


Figura 10.9: Contador de secuencia arbitraria.

```
40   end process;
41
42   co <= '1' when contador = "1001" and enable = '1' else
43         '0';
44   s <= contador;
45 end behavioral;
```



Realice el ejercicio 8

10.9. Contadores de secuencia arbitraria

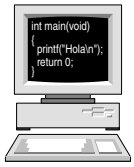
Hasta ahora, los contadores que hemos diseñado cuentan según una secuencia de números naturales consecutivos. No obstante en algunas situaciones se necesitan contadores que cuenten según otra secuencia. Por ejemplo en la introducción se mencionó la existencia de contadores que se saltan el 13 para no herir la sensibilidad de las personas supersticiosas.

En las secciones anteriores hemos visto que los contadores "normales" se implantan como una serie de *flip-flops* tipo T o como un conjunto de *flip-flops* tipo D con un sumador para incrementar (o decrementar) la cuenta. Esto es así precisamente porque estos contadores generan una secuencia de números consecutivos. En los contadores de secuencia arbitraria, la única alternativa para implementarlos es usar una máquina de estados en la que cada valor de la cuenta se representa por un estado.

Por ejemplo, si queremos diseñar un contador ascendente/descendente que cuente según la secuencia 1, 3, 5, 7, 9, la máquina de estados quedaría como la mostrada en la figura 10.9. Dicha máquina dispone de dos entradas: `a_d` para seleccionar el sentido de la cuenta (0 ascendente, 1 descendente) y `enable` para habilitar la cuenta, de forma que cuando esta señal vale cero el contador no cambia de estado y mantiene el valor de la cuenta y cuando vale 1 cambia de estado en sentido ascendente o descendente en función de la señal `a_d`.

Para la implantación física de la máquina de la figura 10.9 existen dos alternativas. La primera es codificar los estados de forma independiente al valor de las salidas, al igual que hemos hecho en todas las máquinas de estados diseñadas en el capítulo 8. En este caso si usamos una codificación de estados binaria, en general se necesitarán menos *flip-flops* para almacenar el estado; pero a cambio será necesario añadir una lógica para generar las salidas del contador a partir del estado actual.

La otra alternativa es codificar el estado con el valor de la cuenta asociado a dicho estado. Por ejemplo, si en el contador que nos ocupa codificamos los estados de forma independiente a las salidas se necesitarán 3 bits, ya que tenemos 5 estados. Si por el contrario hacemos que el estado coincida con el valor binario de la salida necesitaremos cuatro bits.[2]



Realice el ejercicio 9

Descripción en VHDL

En la descripción que vamos a realizar del contador de secuencia arbitraria se van a codificar los estados para que su valor coincida con las salidas. Se deja como ejercicio el codificarla como una máquina de estados con función de salida como las estudiadas en el capítulo 8.

En primer lugar, tal como se puede ver en la línea 19, ahora el estado se almacena en un `std_logic_vector`, que además ha de tener el mismo tamaño que la salida, pues se va a copiar en ella. Aunque no es estrictamente necesario, se han definido también una serie de constantes (líneas 20–24) para que sea más cómoda la especificación de las transiciones entre estados. El resto de la descripción es similar al de las máquinas de estado que hemos visto hasta ahora salvo que no existe el **process** o el **when**--**else** en el que se especifican las salidas, pues ahora éstas son una copia del valor del estado actual.

```
1  -- Contador ascendente/descendente de secuencia arbitraria. La
2  -- secuencia es 1, 3, 5, 7, 9
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6
7  entity ContadorSecArb is
8
9    port (
10     clk     : in  std_logic;
11     reset_n : in  std_logic;
12     a_d     : in  std_logic;
13     enable  : in  std_logic;
14     s       : out std_logic_vector(3 downto 0));
15
16 end ContadorSecArb;
17
18 architecture behavioral of ContadorSecArb is
19   signal estado_act,estado_sig :std_logic_vector(3 downto 0);
20   constant Uno   : std_logic_vector(3 downto 0) := "0001";
21   constant Tres  : std_logic_vector(3 downto 0) := "0011";
```

[2] En este caso particular se puede codificar también con tres bits, usando un pequeño truco que igual al lector espabilado ya se le ha ocurrido.

```
22    constant Cinco : std_logic_vector(3 downto 0) := "0101";
23    constant Siete : std_logic_vector(3 downto 0) := "0111";
24    constant Nueve : std_logic_vector(3 downto 0) := "1001";
25
26  begin  -- behavioral
27
28    VarEstado : process (clk, reset_n, estado_sig)
29    begin
30      if reset_n = '0' then               -- Reset asíncrono
31        estado_act <= Uno;                -- (activo bajo)
32      elsif clk'event and clk = '1' then  -- Flanco de subida
33        estado_act <= estado_sig;
34      end if;
35    end process VarEstado;
36
37    TransicionEstados : process (estado_act, a_d, enable)
38    begin
39      -- Por defecto nos quedamos en el estado actual
40      estado_sig <= estado_act;
41      case estado_act is
42        when Uno =>
43          if enable = '1' then
44            if a_d = '0' then
45              estado_sig <= Tres;
46            else
47              estado_sig <= Nueve;
48            end if;
49          end if;
50        when Tres =>
51          if enable = '1' then
52            if a_d = '0' then
53              estado_sig <= Cinco;
54            else
55              estado_sig <= Uno;
56            end if;
57          end if;
58        when Cinco =>
59          if enable = '1' then
60            if a_d = '0' then
61              estado_sig <= Siete;
62            else
63              estado_sig <= Tres;
64            end if;
65          end if;
```

```
66        when Siete =>
67          if enable = '1' then
68            if a_d = '0' then
69              estado_sig <= Nueve;
70            else
71              estado_sig <= Cinco;
72            end if;
73          end if;
74        when Nueve =>
75          if enable = '1' then
76            if a_d = '0' then
77              estado_sig <= Uno;
78            else
79              estado_sig <= Siete;
80            end if;
81          end if;
82        when others => null;
83      end case;
84    end process;
85
86    s <= estado_act;
87  end behavioral;
```

10.10. Ejercicios

1. Modifique el código del contador binario ascendente mostrado en la sección 10.2 para que el número de bits del contador sea un parámetro genérico.

2. Modifique el código del contador binario descendente mostrado en la sección 10.3 para que el número de bits del contador sea un parámetro genérico.

3. Sustituya en el contador ascendente descendente con habilitación de la cuenta de la sección 10.5 las señales `enable` y `a_d` por las señales `cuenta_arriba` y `cuenta_abajo`. El funcionamiento del contador con estas nuevas señales será el siguiente: en cada flanco de reloj si está activa la señal `cuenta_arriba` el contador se incrementará y si está activa la señal `cuenta_abajo` el contador se decrementará. Cuando ambas señales estén activas a la vez o desactivas, el contador mantendrá su valor.

4. Modifique el contador ascendente módulo 10 de la sección 10.6 para convertirlo en un contador ascendente/descendente módulo 10.

5. Modifique el contador ascendente módulo 10 con acarreo de la sección 10.7 para convertirlo en un contador ascendente/descendente. Tenga en cuenta que existirá una sola salida de acarreo `co` que se activará cuando el contador

rebose de 9 a 0 cuando se esté contando en sentido ascendente o cuando rebose de 0 a 9 cuando se esté contando en sentido descendente.

6. A partir del contador anterior diseñe un contador ascendente/descendente de tres cifras. Use una descripción estructural para instanciar los tres contadores de una cifra y conectar sus acarreos.

7. Diseñe un contador para implementar un reloj. El contador dispondrá de 4 cifras: dos para los segundos y dos para los minutos. De estas 4 cifras, dos de ellas estarán formadas por contadores módulo 10 y las otras dos por contadores módulo 6. Por tanto, el ejercicio ha de dividirse en dos partes.
   - Diseñe en primer lugar un contador ascendente módulo 6.
   - En segundo lugar ha de instanciar los cuatro contadores, dos módulo 10 y dos módulo 6, para implantar el contador de minutos y segundos.

8. Diseñe contador para implementar un temporizador para un microondas. El contador será similar al del ejercicio anterior pero descendente y con carga paralelo para poder inicializar el contador con el tiempo deseado.

9. Especifique el contador de secuencia arbitraria mostrado en la sección 10.9 usando una descripción de la máquina de estados en la que en lugar de especificar la codificación del estado se use una función de salida para obtener el valor de la cuenta.

CAPÍTULO 11

Diseño de sistemas complejos: ruta de datos + control

En este capítulo estudiaremos la metodología para diseñar sistemas digitales complejos a partir de los elementos básicos que hemos estudiado en los capítulos anteriores. Veremos en primer lugar la importancia del diseño jerárquico y en segundo lugar cómo dividir el diseño en dos partes bien diferenciadas: el camino de datos que se encarga de procesar los datos y el control que se encarga de controlar el camino de datos.

11.1. Introducción

El diseño de sistemas digitales complejos requiere una metodología que nos permita abordar dicha complejidad. Como ya se expuso en la sección 4.2 el primer paso en el diseño de un sistema es su división en bloques jerárquicos de forma que conforme se baja en la jerarquía los bloques son cada vez más simples, terminando cuando éstos son lo suficientemente simples como para implementarlos con unos pocos elementos, o con unas pocas líneas de código si trabajamos con lenguajes de descripción de *hardware*.

Dentro de esta división es conveniente separar las dos partes de las que se componen casi todos los sistemas digitales, las cuales se ilustran en la figura 11.1. Por

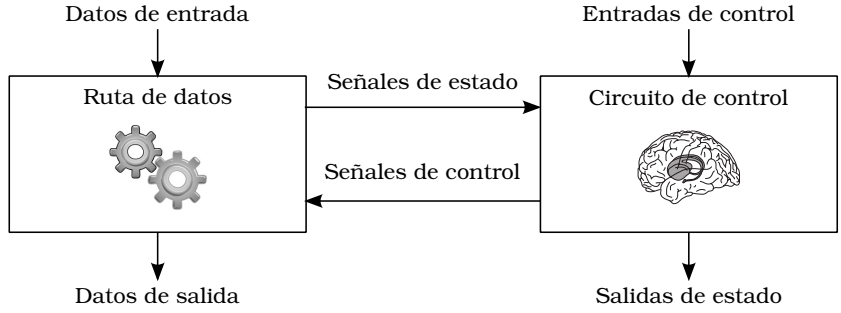


Figura 11.1: Descomposición de un sistema digital en ruta de datos + control.

un lado existe una parte encargada de encaminar los datos por el circuito, realizando ciertos procesos sobre ellos. A esta parte se le denomina ruta de datos, camino de datos o *datapath* en inglés. El camino de datos está compuesto por registros donde se almacenan los datos, circuitos aritméticos donde se hacen operaciones con los datos, multiplexores para regular el trasiego de datos entre bloques, contadores, etc. Podemos pensar en ella como el lugar en el que se hace el "trabajo duro" del sistema. Para ello, la ruta de datos contendrá una serie de señales para controlarla. Por ejemplo los registros tendrán señales de `enable` para habilitarlos, los multiplexores tienen entradas de selección, etc. Por otro lado la ruta de datos también dispondrá de una serie de señales que nos informan sobre el su estado. Por ejemplo, los contadores nos informarán del fin de una cuenta, los sumadores si ha ocurrido un desbordamiento, etc.

La segunda parte consiste en un circuito que mediante la lectura de las señales de estado y el manejo de las señales de control de la ruta de datos es capaz de conseguir que el circuito realice su función. Este circuito también necesita señales externas, como por ejemplo un pulsador que le indique que tiene que comenzar su trabajo o un led con el que indique si se ha producido un error. Se puede decir por tanto que este circuito es el que aporta la "inteligencia" al sistema digital y normalmente se construye mediante una máquina de estados.

Lamentablemente no hay una metodología rígida para realizar el proceso de diseño, por lo que en este capítulo vamos a introducirla mediante el estudio de una serie de ejemplos que ilustrarán como enfrentarse al diseño de un sistema digital complejo.

11.2. Control de una barrera de aparcamiento

En esta sección vamos a diseñar un circuito para controlar una barrera de aparcamiento automática como la mostrada en la figura 11.2. Dicha barrera se controla mediante cuatro señales. Mediante `sube` y `baja` se activa el motor de la barrera para subirla y bajarla respectivamente. En la barrera se han situado dos fines de carrera que se ponen a 1 cuando la barrera está completamente subida (`fc_arr`) o cuando está completamente bajada (`fc_abj`). Delante de la barrera se ha situado de un sensor inductivo situado en el suelo cuya salida `det_coche` se pone a 1 cuando se sitúa un coche encima de él.

El sistema abrirá automáticamente la barrera cuando detecte un coche. El funcionamiento del sistema ha de ser el siguiente:

- Cuando se detecte un coche ha de subirse la barrera. Para ello el circuito ha de activar la señal `sube` y mantenerla así hasta que la barrera termine de subir, en cuyo momento se desactivará la señal `sube` para no quemar el motor. La detección de que la barrera ha llegado arriba se realizará mediante el fin de carrera `fc_arr`, que como se ha dicho antes se pondrá a 1 cuando la barrera llegue arriba.
- Mientras se siga detectando el coche la barrera permanecerá subida. Cuando deje de detectarse se esperarán 5 segundos antes de bajarla. Si mientras se

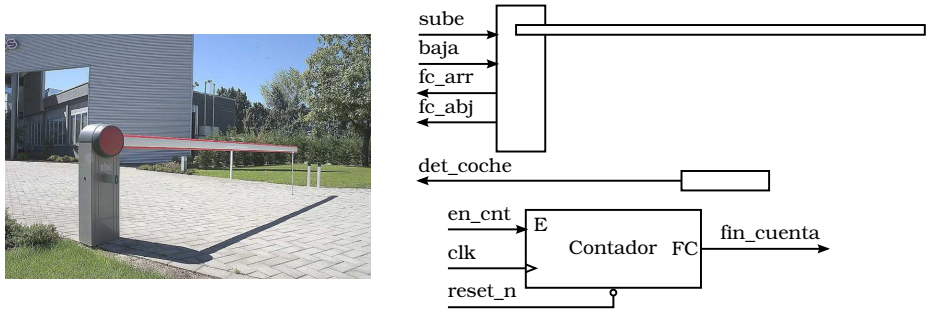


Figura 11.2: Barrera de aparcamiento.

están esperando estos 5 segundos se vuelve a detectar otro coche se abortará la espera.

- Para bajar la barrera se realizará el proceso inverso al seguido para subirla: se activará la señal `baja` hasta detectar la activación del fin de carrera `fc_abj`.
- Si mientras se está bajando la barrera se detecta un coche se volverá a subir.

11.2.1. Camino de datos

El camino de datos en este caso es muy sencillo, pues consta tan solo de un contador para realizar la temporización de cinco segundos y de las señales de entrada y salida para controlar la barrera, tal como puede verse en la figura 11.2.

Si suponemos que el reloj del sistema es de 50 MHz, necesitaremos un contador que cuente hasta:

$$cuenta = 5\ s \cdot 50\,000\,000\ Hz = 250\,000\,000$$

Como:

$$log_2(cuenta) = \log_2(250\,000\,000) = 27{,}89$$

se necesitarán 28 bits para almacenar la cuenta.

El contador permanecerá a cero mientras la señal `en_cnt` sea cero. Cuando se ponga a 1 el contador contará en cada ciclo de reloj y cuando llegue a su valor final (250.000.000) se activará la señal `fin_cuenta`. De esta forma el circuito de control podrá arrancar el contador mediante la señal de *enable* (`en_contador`) cuando quiera comenzar la espera de 5 segundos y sabrá que esta espera ha terminado cuando se active la señal `fin_cnt`. Cuando se tenga que abortar la cuenta bastará con desactivar la señal de *enable*.

11.2.2. Circuito de control

El circuito de control de la barrera será el encargado de controlar dicha barrera y el contador para conseguir el funcionamiento deseado. Para ello lo más fácil es

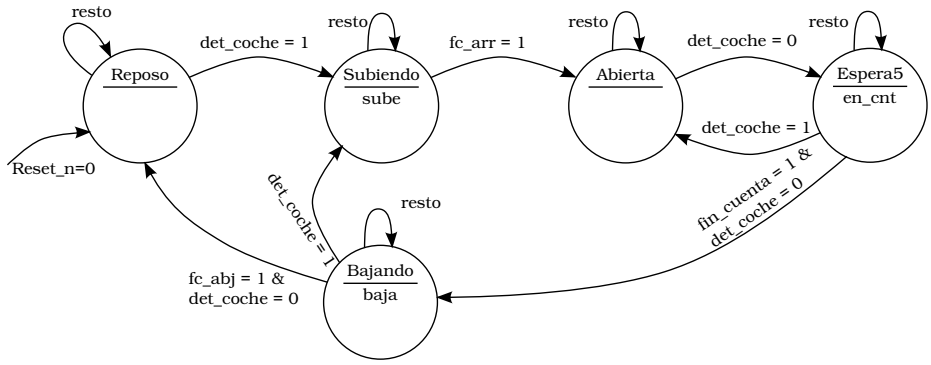


Figura 11.3: Diagrama de estados del circuito de control para la barrera de aparcamiento.

implementarlo con una máquina de estados, cuyo diagrama se muestra en la figura 11.3. En este diagrama se ha usado una notación menos formal que la usada en el capítulo 8 aunque más práctica cuando tenemos un gran número de señales. En ella se ha puesto en cada transición sólo la entrada o combinación de entradas que la activa. El resto de combinaciones de entradas hace que la máquina permanezca en el mismo estado, lo cual se representa con la transición al mismo estado con la palabra "resto". De la misma forma en cada estado sólo se han representado las salidas que se ponen a uno en dicho estado, limitándonos a poner su nombre. El resto de salidas permanecerán a cero en dicho estado.

Como se puede ver en la figura 11.3, el comportamiento del circuito de control es el especificado en el apartado anterior. Cuando se detecte un coche se pasará al estado Subiendo, donde se subirá la barrera. En este estado permaneceremos hasta que se active el fin de carrera `fc_arr` que detecta que la barrera ha subido por completo; pasando al estado Abierta. En este estado permaneceremos hasta que deje de detectarse el coche. En este momento pasaremos al estado Espera5, donde arrancaremos el contador y esperaremos a que nos indique mediante su salida `fin_cuenta` que han pasado los 5 segundos. Si antes de que esto ocurra se vuelve a detectar otro coche se pasará de nuevo al estado Abierta, pues es necesario mantener la puerta abierta hasta que termine de pasar el coche si no queremos causarle graves perjuicios a su dueño. Si no se detecta un coche pasados los 5 segundos, pasamos al estado Bajando para bajar la barrera. Si se detecta un coche en este estado pasaremos al estado Subiendo para darle paso. Si no se detecta y vemos que la barrera ha llegado abajo pasamos al estado de Reposo a esperar un nuevo coche.

11.2.3. Descripción en VHDL

La descripción del circuito se ha realizado en un único archivo VHDL, ya que la ruta de datos es tan sencilla que no merece la pena implementarla aparte. No obstante se han separado ambas secciones del archivo claramente mediante comentarios para que no haya duda de la división entre ruta de datos y control.

El contador de la ruta de datos se ha implementado mediante un **process** y la única diferencia respecto a los contadores de la sección anterior es que cuando se deshabilita el contador vuelve a 0, tal como se muestra en la línea 41.

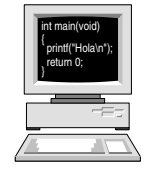
El circuito de control se ha implementado mediante una máquina de estados. Nótese que el código, tanto de las transiciones como de las salidas, es un reflejo de la notación simplificada de la figura 11.3. Así, en las transiciones cada flecha se ha traducido por un **if** y en el **process** de las salidas en cada **when** aparecen sólo las salidas presentes en cada estado del diagrama.

```
1  -- Circuito para controlar una barrera automática para
2  -- aparcamiento
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  entity BarreraAparcamiento is
9
10   port (
11     clk       : in  std_logic;
12     reset_n   : in  std_logic;
13     fc_arr    : in  std_logic;          -- Fin carrera arriba
14     fc_abj    : in  std_logic;          -- Fin carrera abajo
15     det_coche : in  std_logic;          -- detector de coches
16     sube      : out std_logic;          -- Sube la barrera
17     baja      : out std_logic);         -- Baja la barrera
18
19 end BarreraAparcamiento;
20
21 architecture behavioral of BarreraAparcamiento is
22   signal contador : unsigned(27 downto 0);  -- Contador 5 s
23   signal en_cnt : std_logic;         -- Enable del contador
24   signal fin_cuenta : std_logic;
25
26   type t_estados is (Reposo, Subiendo, Abierta, Espera5,
27                      Bajando);
28   signal estado_act, estado_sig : t_estados;
29 begin  -- behavioral
30   ------------------------------------------------------------
```

```
31  -- Camino de datos.
32  ----------------------------------------------------------------
33  Conta5Seg: process (clk, reset_n)
34  begin  -- process Conta5Seg
35    if reset_n = '0' then
36      contador <= (others => '0');
37    elsif clk'event and clk = '1' then
38      if en_cnt = '1' then
39        contador <= contador + 1;
40      else
41        contador <= (others => '0');
42      end if;
43    end if;
44  end process Conta5Seg;
45  fin_cuenta <= '1' when contador = 249999999 else
46                '0';
47
48  ----------------------------------------------------------------
49  --  Circuito de control
50  ----------------------------------------------------------------
51  VarEstado : process (clk, reset_n)
52  begin
53    if reset_n = '0' then                -- Reset asíncrono
54      estado_act <= Reposo;              -- (activo bajo)
55    elsif clk'event and clk = '1' then   -- Flanco de subida
56      estado_act <= estado_sig;
57    end if;
58  end process VarEstado;
59
60  TransicionEstados : process (estado_act, fc_arr, fc_abj,
61                               det_coche, fin_cuenta)
62  begin
63    -- Por defecto nos quedamos en el estado actual
64    estado_sig <= estado_act;
65    case estado_act is
66      when Reposo =>
67        if det_coche = '1' then
68          estado_sig <= Subiendo;
69        end if;
70      when Subiendo =>
71        if fc_arr = '1' then
72          estado_sig <= Abierta;
73        end if;
74      when Abierta =>
```

```
75          if det_coche = '0' then
76            estado_sig <= Espera5;
77          end if;
78        when Espera5 =>
79          if det_coche = '1' then
80            estado_sig <= Abierta;
81          elsif fin_cuenta = '1' then
82            estado_sig <= Bajando;
83          end if;
84        when Bajando =>
85          if det_coche = '1' then
86            estado_sig <= Subiendo;
87          elsif fc_abj = '1' then
88            estado_sig <= Reposo;
89          end if;
90        when others =>
91          estado_sig <= Reposo;
92      end case;
93    end process;
94
95    Salidas: process (estado_act)
96    begin  -- process Salidas
97      sube <= '0';
98      baja <= '0';
99      en_cnt <= '0';
100     case estado_act is
101       when Reposo =>
102         null;
103       when Subiendo =>
104         sube <= '1';
105       when Abierta =>
106         null;
107       when Espera5 =>
108         en_cnt <= '1';
109       when Bajando =>
110         baja <= '1';
111       when others => null;
112     end case;
113   end process Salidas;
114 end behavioral;
```

Una simulación del circuito se muestra en la figura 11.4. En ella se ha cambiado el valor de la cuenta a 25 para poder realizar la simulación en un tiempo razonable y poder visualizarla mejor.



Realice el ejercicio 1

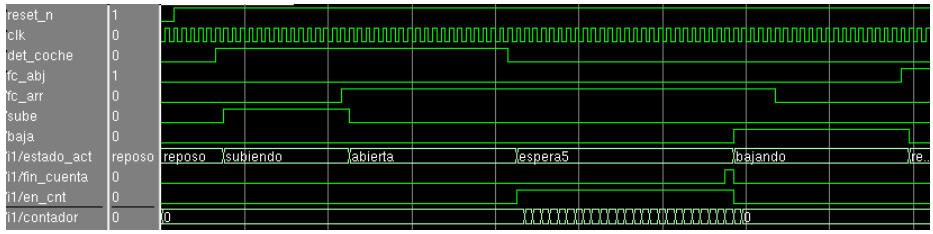


Figura 11.4: Simulación de la barrera de aparcamiento.

11.3. Control de calidad de toros

En esta sección vamos a diseñar un circuito que nos han pedido los organizadores de los festejos taurinos de Navas del Olivo. Dicho circuito ha de ser capaz de medir la velocidad de los toros antes de adquirirlos. Para ello han ideado un sistema que consiste en un pasillo de 10 metros de longitud en cuyo inicio y fin se han situado dos células fotoeléctricas que se activan al paso del toro, tal como se muestra en la figura 11.5. Para medir la velocidad del toro se colocará a éste al principio del pasillo y desde el final se le enseñará un capote para que salga corriendo en su busca. El circuito ha de medir el tiempo que emplea en recorrer los 10 m y a partir de éste calcular su velocidad media. Para medir este tiempo, se arrancará un temporizador cuando se active la primera célula (señal `cel_ini`) y se parará cuando llegue al final (señal `cel_fin`). Mientras se esté midiendo el tiempo se activará la salida `midiendo` para indicarlo. Una vez medido el tiempo, si la velocidad media supera los 20 km/h se dará el toro por bueno y se activará la señal `toro_bueno`. Si por el contrario la velocidad es menor o igual a los 20 km/h se activará la señal `toro_malo` para que dicho toro sea desechado. Ambas señales han de mantenerse activas hasta que se de el reset al circuito o hasta que pase otro toro por la célula inicial.

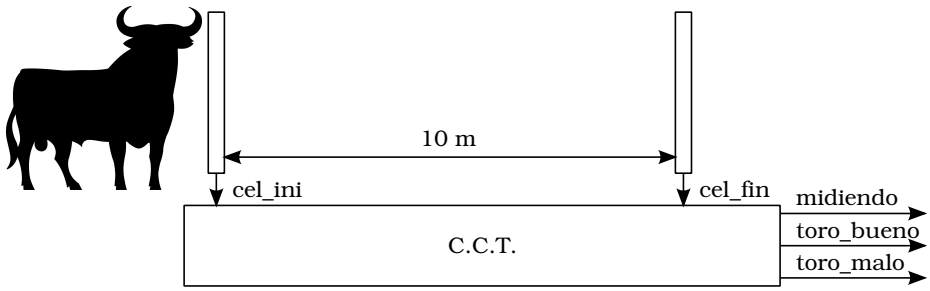


Figura 11.5: Diagrama de bloques del control de calidad de toros (C.C.T.).

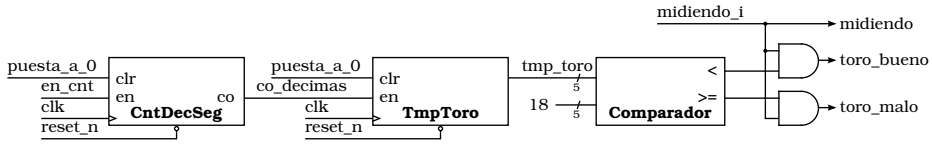


Figura 11.6: Ruta de datos del control de calidad de toros.

11.3.1. Ruta de datos

Pensando un poco el problema anterior, lo único que necesitamos hacer es calcular el tiempo que emplearía un toro en recorrer los 10 metros del pasillo si fuese a 20 km/h de media y comparar ese tiempo con el empleado por el toro cuya calidad se desea comprobar. Si el tiempo es menor el toro será bueno y si es mayor o igual el toro será malo.

Para calcular el tiempo, teniendo en cuenta la ecuación del movimiento rectilíneo y uniforme se obtiene:

$$v = \frac{s}{t} \Rightarrow t = \frac{s}{v} = \frac{10m}{20\,km/h} = \frac{10\,m}{20\,000\,m/3600\,s} = \frac{10 \cdot 3\,600\,s}{20\,000} = 1,8\,s$$

Así pues para la ruta de datos del circuito necesitamos en primer lugar un contador para poder medir el tiempo empleado por el toro en recorrer los 10 metros y en segundo lugar un comparador para comparar dicho tiempo con $1,8$ segundos. Para evitar trabajar con números decimales, lo más cómodo es realizar un contador que cuente cada $0,1$ segundos y comparar dicha cuenta con 18. El número de bits del contador tendrá que ser por tanto igual a 5, para poder así representar el 18. No obstante se ha de evitar que el contador rebose, pues si no lo hacemos, como el contador pasará de 31 a 0, podemos dar por bueno un toro que tarde $3,2$ segundos en realizar su carrera. Por último, hemos de ser capaces de poner a cero el contador cuando el toro pase por la primera foto-célula y de pararlo cuando pase por la segunda. De esta forma podremos mantener el estado de las salidas `toro_bueno` y `toro_malo` hasta que se vuelva a realizar otra medida.

Para conseguir que el contador anterior cuente cada décima de segundo se necesita otro contador, el cual generará un acarreo de salida cuando la cuenta llegue a:

$$cuenta = 0,1\ s \cdot 50\,000\,000\ Hz = 5\,000\,000$$

Como:

$$log_2(cuenta) = \log_2(5\,000\,000) = 22,25$$

y por tanto se necesitarán 23 bits para almacenar la cuenta.

Al igual que con el contador anterior, se necesita poner también este contador a cero cuando el toro pase por la primera fotocélula para así realizar una medida pre-

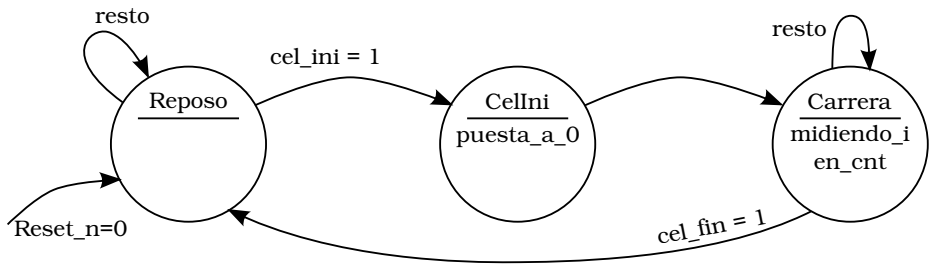


Figura 11.7: Diagrama de estados del circuito de control de calidad de toros.

cisa. De lo contrario, tendríamos una imprecisión de ±0,05 segundos, dependiendo de dónde estuviese este contador cuando el toro pasa por la primera célula.[1]

Por último, la especificación del problema requiere que las salidas `toro_bueno` y `toro_malo` estén inactivas mientras dure la medida. Como ambas se obtienen de la salida del comparador es necesario usar una puerta AND para bloquearlas. Afortunadamente la propia señal `midiendo` que se activa durante la medida nos sirve para realizar el bloqueo. En la figura 11.6 se muestra la ruta de datos completa. Tan solo destacar para terminar que el hecho de necesitar la señal `midiendo` internamente en el código VHDL nos obliga a disponer de una copia a la que se ha denominado `midiendo_i`, la cual se ha incluido en el diagrama para una mayor igualdad entre el diagrama y el código.

11.3.2. Circuito de control

El circuito de control es un poco más simple que el de la barrera de aparcamiento, tal como se muestra en el diagrama de estados de la figura 11.7. Cuando se activa `cel_ini` se pasa al estado `CelIni` donde se activa la señal `puesta_a_0` para poner a cero los temporizadores. De este estado se pasa a `Carrera` donde se activa la señal `midiendo_i` y se da el enable a los contadores. Cuando el toro llegue al final de la carrera y se active la señal `cel_fin` se pasa al estado de `Reposo`, donde se desactiva el enable de los contadores y la señal `midiendo_i`. Al haber quitado el enable de los contadores, éstos mantienen su valor, con lo que en las salidas del comparador tendremos si el tiempo empleado ha sido menor de 1,8 segundos y por tanto el toro es bueno o viceversa. Al haber desactivado la señal `midiendo_i` el resultado del comparador se mostrará en las salidas hasta que salgamos de dicho estado.

[1] Otra alternativa de diseño a la propuesta es construir un temporizador similar al diseñado en la sección anterior que incrementase su cuenta en cada ciclo de reloj y comparar la salida del dicho temporizador con $1{,}8\,s \cdot 50MHz = 90 \cdot 10^6$. Aunque la solución también es válida, para comparar el valor del temporizador con $90 \cdot 10^6$ necesitaremos un comparador de 27 bits, ya que $\log_2(90 \cdot 10^6) = 26{,}42$. Obviamente la solución propuesta, al necesitar sólo un comparador de 5 bits, necesita menos *hardware*.

11.3.3. Descripción en VHDL

A continuación se muestra el código en VHDL del circuito diseñado. Como puede apreciar el código es un reflejo de la ruta de datos y del circuito de control mostrados en las secciones anteriores.

```
1  -- Circuito para realizar un control de calidad de toros
2  -- midiendo su velocidad media de carrera durante 10 m.
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  entity ControlCalidadToros is
9
10   port (
11     clk              : in  std_logic;
12     reset_n          : in  std_logic;
13     cel_ini, cel_fin : in  std_logic;   -- células fotoeléct.
14     midiendo         : out std_logic;
15     toro_bueno       : out std_logic;
16     toro_malo        : out std_logic);
17
18 end ControlCalidadToros;
19
20 architecture behavioral of ControlCalidadToros is
21   -- Contador de décimas de segundo
22   signal cnt_dec_seg : unsigned(22 downto 0);
23   -- Una décima de segundo para un reloj de 50 MHz
24   constant una_decima : unsigned(22 downto 0)
25     := to_unsigned(4999999,23);    -- poner a 4 para simular
26   -- Acarreo de salida del contador de décimas
27   signal co_decimas : std_logic;
28   -- Enable para el contador de décimas de segundo
29   signal en_cnt : std_logic;
30   -- Pone a cero el contador de décimas de segundo y
31   -- el temporizador del toro
32   signal puesta_a_0 : std_logic;
33   -- Tiempo de carrera del toro en décimas de segundo.
34   signal tmp_toro : unsigned(4 downto 0);
35   signal midiendo_i : std_logic;
36   -- Salidas comparador
37   signal tmp_mayor_ig_18, tmp_menor_18 : std_logic;
38   type t_estados is (Reposo, CelIni, Carrera);
39   signal estado_act, estado_sig : t_estados;
```

```
40 begin  -- behavioral
41
42 ----------------------------------------------------------------
43 -- Ruta de datos
44 ----------------------------------------------------------------
45   -- Contador de décimas de segundo
46   CntDecSeg: process (clk, reset_n)
47   begin  -- process CntDecSeg
48     if reset_n = '0' then
49       cnt_dec_seg <= (others => '0');
50     elsif clk'event and clk = '1' then
51       if puesta_a_0 = '1' then
52         cnt_dec_seg <= (others => '0');
53       elsif en_cnt = '1' then
54         if cnt_dec_seg = una_decima then
55           cnt_dec_seg <= (others => '0');
56         else
57           cnt_dec_seg <= cnt_dec_seg + 1;
58         end if;
59       end if;
60     end if;
61   end process CntDecSeg;
62   co_decimas <= '1' when cnt_dec_seg = una_decima
63                          and en_cnt = '1' else
64                 '0';
65
66   -- Temporizador para medir el tiempo empleado por el
67   -- toro en su carrera.
68   TmpToro: process (clk, reset_n)
69   begin  -- process TmpToro
70     if reset_n = '0' then
71       tmp_toro <= (others => '0');
72     elsif clk'event and clk = '1' then
73       if puesta_a_0 = '1' then
74         tmp_toro <= (others => '0');
75       elsif co_decimas = '1' and tmp_toro /= "11111" then
76         -- El contador no ha de rebosar.
77         tmp_toro <= tmp_toro + 1;
78       end if;
79     end if;
80   end process TmpToro;
81
82   -- Comparador con 18 décimas de segundo.
83   tmp_mayor_ig_18 <= '1'when tmp_toro >= 18 else
```

```
84                      '0';
85    tmp_menor_18 <= '1' when tmp_toro < 18 else
86                    '0';
87
88    -- Leds de salida
89    toro_bueno <=  tmp_menor_18 and (not midiendo_i);
90    toro_malo <= tmp_mayor_ig_18 and (not midiendo_i);
91    midiendo <= midiendo_i;
92
93  ------------------------------------------------------------
94  -- Circuito de control
95  ------------------------------------------------------------
96    VarEstado : process (clk, reset_n)
97    begin
98      if reset_n = '0' then               -- Reset asíncrono
99        estado_act <= Reposo;             -- (activo bajo)
100     elsif clk'event and clk = '1' then  -- Flanco de subida
101       estado_act <= estado_sig;
102     end if;
103   end process VarEstado;
104
105   TransicionEstados : process (estado_act, cel_ini, cel_fin)
106   begin
107     -- Por defecto nos quedamos en el estado actual
108     estado_sig <= estado_act;
109     case estado_act is
110       when Reposo =>
111         if cel_ini = '1' then
112           estado_sig <= CelIni;
113         end if;
114       when CelIni =>
115         estado_sig <= Carrera;
116       when Carrera =>
117         if cel_fin = '1' then
118           estado_sig <= Reposo;
119         end if;
120       when others =>
121         estado_sig <= Reposo;
122     end case;
123   end process;
124
125   Salidas: process (estado_act)
126   begin  -- process Salidas
127     midiendo_i <= '0';
```

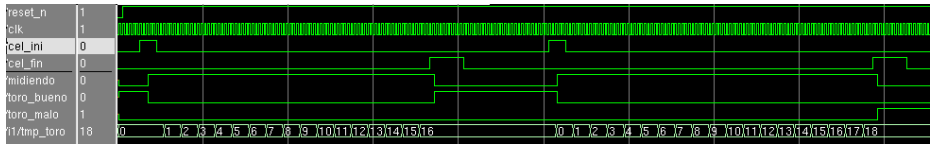
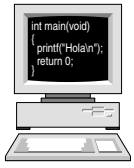


Figura 11.8: Simulación del control de calidad de toros.

```
128     puesta_a_0 <= '0';
129     en_cnt <= '0';
130     case estado_act is
131       when Reposo =>
132         null;
133       when CelIni =>
134         puesta_a_0 <= '1';
135       when Carrera =>
136         midiendo_i <= '1';
137         en_cnt <= '1';
138       when others => null;
139     end case;
140   end process Salidas;
141
142 end behavioral;
```



Realice el ejercicio 2

Una simulación del circuito se muestra en la figura 11.8. En ella se ha cambiado el valor de la contador de décimas a 4 para poder realizar la simulación en un tiempo razonable y poder visualizarla mejor.

11.4. Conversor de binario a BCD

En los sistemas digitales se suele trabajar en binario, ya que como se ha visto en los capítulos anteriores es mucho más eficiente hacer las operaciones en binario, tanto desde el punto de vista de la rapidez de los cálculos como desde el *hardware* necesario. Sin embargo, la visualización de los números en binario es tediosa para nosotros, acostumbrados como estamos a trabajar en decimal. Por ello, cuando es necesario visualizar un número binario en un *display* es conveniente realizar su conversión a BCD para visualizarlo en decimal. En esta sección se va a mostrar un algoritmo para realizarlo que es fácil de implantar en *hardware* usando registros de desplazamiento.

11.4.1. El algoritmo Double-Dabble

Este algoritmo comienza escribiendo el número binario a la derecha y reservando espacio a la izquierda para los dígitos en BCD que serán necesarios según el número

de bits del número binario. Por ejemplo, si queremos convertir el número binario 1100 (12), empezaríamos escribiendo:

Dec.	Unid.	Bin.
		1100

El algoritmo se basa en desplazar tantas veces como bits tengamos para ir introduciendo los bits del número binario en las posiciones reservadas para los dígitos en BCD. Así, tras los tres primeros desplazamientos tendremos:

Dec.	Unid.	Bin.
		1100
	1	100
	11	00
	110	0

Que de momento no indican ningún peligro, ya que tanto las decenas como las unidades contienen dígitos BCD válidos. El problema viene al realizar el cuarto desplazamiento, ya que el dígito de las unidades se convierte en un dígito inválido:

Dec.	Unid.	Bin.
		1100
	1	100
	11	00
	110	0
	1100	

Para arreglar semejante desaguisado habría que sumar 6 y acarrear (es decir sumar) uno a la siguiente cifra. No obstante el acarreo a la siguiente cifra nos lo podemos ahorrar si nos damos cuenta que al desplazar a la izquierda estamos multiplicando por dos. Teniendo esto en cuenta, si en lugar de desplazar y sumar 6 si vemos que el dígito BCD contiene un valor mayor o igual a 10, sumamos 3 si el dígito tiene un valor mayor o igual a 5 y luego desplazamos; matemáticamente estamos haciendo lo mismo. La diferencia práctica de la segunda opción sin embargo es considerable. Si tenemos en cuenta que 5+3 = 8 y el 8 en binario es 1000, cuando realicemos el desplazamiento, el 1 del 1000 va a pasar a la siguiente cifra, con lo que nos estamos ahorrando el acarreo. Con el resto de números nos ocurrirá lo mismo al sumar 3: el acarreo nos saldrá gratis. Por tanto el algoritmo queda:

Acción	Dec.	Unid.	Bin.
			1100
Despl.		1	100
Despl.		11	00
Despl.		110	0
Suma 3		1001	0
Despl.	1	0010	

Que como puede comprobar es el número 12 en BCD, con lo que el algoritmo funciona a las mil maravillas.

En resumen el algoritmo consiste en lo siguiente:

- Escriba el número a convertir a la derecha y a su izquierda reserve el espacio necesario para las cifras en BCD que se necesiten.
- Repita tantas veces como bits tenga el número binario:
  - Si el resultado en alguno de los dígitos BCD es mayor que 4,[2] se le sumará 3 a dicho dígito.
  - Desplace un bit a la izquierda.

Tenga en cuenta que en el resultado final pueden existir números mayores que 4, por lo que en el algoritmo se ha puesto primero la suma y luego el desplazamiento. Obviamente, como al principio todos los dígitos BCD estarán a cero no se sumará ningún 3.

Para ilustrar el algoritmo nada mejor que ver otro ejemplo. Vamos a convertir ahora el número de 8 bits 254 (1111 1110):

Acción	Cent.	Dec.	Unid.	Bin.
				1111 1110
Despl.			1	1111 110
Despl.			11	1111 10
Despl.			111	1111 0
Suma 3 Unid.			1010	1111 0
Despl.		1	0101	1110
Suma 3 Unid.		1	1000	1110
Despl.		11	0001	110
Despl.		0110	0011	10
Suma 3 Dec.		1001	0011	10
Despl.	1	0010	0111	0
Suma 3 Unid.	1	0010	1010	0
Despl.	10	0101	0100	
Resultado	2	5	4	

11.4.2. Camino de datos

La ruta de datos se muestra en la figura 11.9, y consta de los siguientes elementos:

- Un contador de 3 bits (CntBits) para contar el número de desplazamientos realizador. El contador se pone a cero por el circuito de control en el inicio de la conversión y se incrementa en cada desplazamiento, para lo que se usan las señales `puesta_a_0` y `en_cnt` respectivamente. Cuando el contador llegue a 7 (habrá contado 8 veces pues empieza en cero) activa la señal `fin_cnt`.

[2] Tenga en cuenta que es lo mismo comprobar si el número es mayor o igual que 5 que mayor que 4.

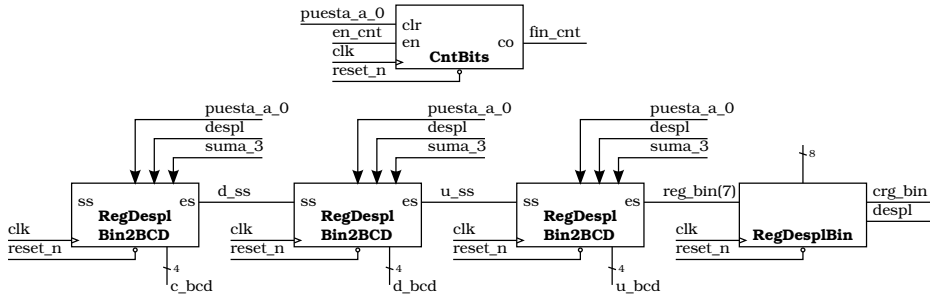


Figura 11.9: Ruta de datos del conversor de binario a BCD.

- Un registro de desplazamiento de entrada paralelo salida serie denominado RegDesplBin en el que se carga el número en binario (mediante la señal `crg_bin`) y luego se desplaza para realizar la conversión (mediante `Despl`).

- Tres registros de desplazamiento de entrada serie salida paralelo denominados RegDesplBin2BCD. Estos registros disponen de tres entradas de control: `puesta_a_0` para ponerlos a cero al inicio de la conversión, `despl` para desplazar un bit a la izquierda y `suma_3` que hace que el circuito sume 3 a su contenido si éste es mayor que 4. En caso contrario la señal `suma_3` no hará nada.

Los cuatro registros se han conectado en serie para que la salida serie de cada uno de ellos sea la entrada del registro situado a su izquierda.

11.4.3. Circuito de control

El diagrama de estados del circuito de control se muestra en la figura 11.10. El circuito espera en el estado de reposo la activación de la entrada `conv` que es la que arranca el proceso de conversión. En este momento se pasa al estado Carga donde se carga el valor presente en la entrada paralela `n_bin` en el registro de desplazamiento RegDesplBin y se ponen a cero tanto el contador como los registros de desplazamiento BCD. A continuación se pasa al estado Desplaza, donde se realiza un desplazamiento y se incrementa la cuenta del contador de bits procesados. De este estado se pasará a Suma3 para indicarle a los registros de desplazamiento BCD que sumen 3 al número que contienen si éste es mayor que 4. Del estado Suma3 se volverá al estado Desplaza, repitiéndose el ciclo 8 veces para desplazar los 8 bits de la entrada. Para detectar el final de la conversión se usa la señal `fin_Cnt`, que es puesta a 1 por el contador cuando ha contado 8 veces. En este último caso se pasa al estado FinConv donde se activa la señal `hecho` para indicar que la conversión ha finalizado y se espera a que baje la señal `conv` si aún no lo ha hecho para evitar realizar una nueva conversión que no se ha solicitado.

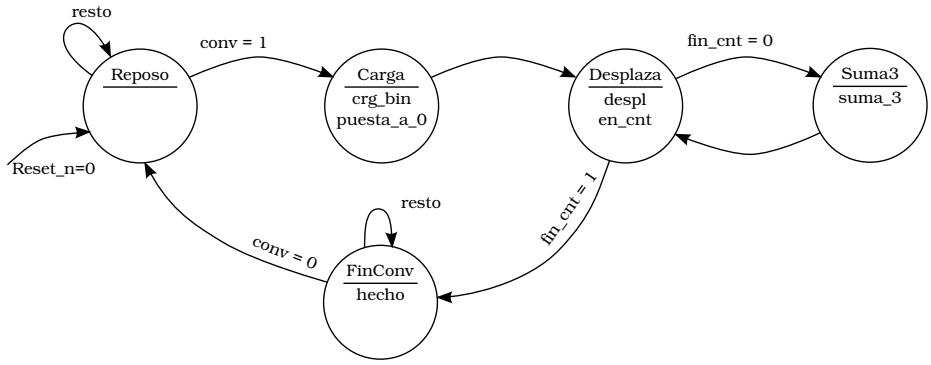


Figura 11.10: Diagrama de estados del conversor de binario a BCD.

11.4.4. Descripción en VHDL

La descripción en VHDL se ha dividido en dos partes. En primer lugar se ha diseñado un registro de desplazamiento de 4 bits para albergar un dígito BCD:

```
1  -- Registro de desplazamiento para un dígito en BCD.
2  -- Forma parte del conversor de binario a BCD.
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  entity RegDesplBin2BCD is
9
10   port (
11     clk     : in  std_logic;
12     reset_n : in  std_logic;
13     es      : in  std_logic;                 -- Entrada serie
14     despl   : in  std_logic;                -- Desplazamiento
15     suma_3  : in  std_logic;           -- Suma 3 (si procede)
16     pst_a_0 : in std_logic;                      -- Puesta a 0
17     n_bcd   : out std_logic_vector(3 downto 0); -- Salida BCD
18     ss      : out std_logic);                -- Salida serie
19
20 end RegDesplBin2BCD;
21
22 architecture behavioral of RegDesplBin2BCD is
23   signal n_bcd_i : unsigned(3 downto 0);  -- bcd interno
24 begin  -- behavioral
25
```

```
  RegDespl : process (clk, reset_n)
  begin  -- process RegDespl
    if reset_n = '0' then
      n_bcd_i <= (others => '0');
    elsif clk'event and clk = '1' then
      if pst_a_0 = '1' then
        n_bcd_i <= (others => '0');
      elsif despl = '1' then               -- Despl. a izq.
        n_bcd_i(3 downto 1) <= n_bcd_i(2 downto 0);
        n_bcd_i(0)          <= es;
      elsif suma_3 = '1' and n_bcd_i > 4 then
        -- En el ciclo que toca sumar, óslo se suma 3 si
        -- el dígito es mayor que 4.
        n_bcd_i <= n_bcd_i + 3;
      end if;
    end if;
  end process RegDespl;

  -- Asignación de salidas
  n_bcd <= std_logic_vector(n_bcd_i);
  ss <= n_bcd_i(3);            -- La salida serie es el MSB
end behavioral;
```

En segundo lugar se ha descrito el resto de la ruta de datos, la cual instancia tres registros de desplazamiento de los descritos en el archivo anterior. En el mismo archivo también se describe la máquina de estados de control. Fíjese que tanto la ruta de datos como el circuito de control son un reflejo de las figuras 11.9 y 11.10

```
-- Conversor de binario a BCD usando el algoritmo
-- "double-dabble"

library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity ConvBin2BCD is

  port (
    clk     : in  std_logic;
    reset_n : in  std_logic;
    n_bin   : in  std_logic_vector(7 downto 0);  -- Entr. bin
    conv    : in  std_logic;            -- Arranca conversión
    hecho   : out std_logic;            -- Fin conversión
    u_bcd   : out std_logic_vector(3 downto 0);  -- Unid. BCD
    d_bcd   : out std_logic_vector(3 downto 0);  -- Dec.  BCD
```

```
18      c_bcd   : out std_logic_vector(3 downto 0)); -- Cent. BCD
19
20  end ConvBin2BCD;
21
22  architecture behavioral of ConvBin2BCD is
23    --  Reg Despl para binario
24    signal reg_bin : unsigned(7 downto 0);
25    signal crg_bin : std_logic;   -- Carga paralelo reg bin
26    signal despl : std_logic;     -- Desplazamiento
27    -- Suma 3 a los dígitos que lo necesiten
28    signal suma_3 : std_logic;
29    -- Salidas serie de los Registros de desplazamiento BCD
30    signal u_ss, d_ss : std_logic;
31    -- Cuenta los bits que han sido ya procesados
32    signal cnt_bits : unsigned(2 downto 0);
33    signal en_cnt : std_logic;            -- Enable del cnt
34    signal fin_cnt : std_logic;           -- Fin de cuenta (7)
35    signal puesta_a_0 : std_logic;      -- Puesta a 0 reg y cnt
36    type t_estados is (Reposo, Carga, Desplaza, FinConv, Suma3);
37    signal estado_act, estado_sig : t_estados;
38
39    component RegDesplBin2BCD
40      port (
41        clk     : in  std_logic;
42        reset_n : in  std_logic;
43        es      : in  std_logic;
44        despl   : in  std_logic;
45        suma_3  : in  std_logic;
46        pst_a_0 : in  std_logic;
47        n_bcd   : out std_logic_vector(3 downto 0);
48        ss      : out std_logic);
49    end component;
50
51  begin  -- behavioral
52  ------------------------------------------------------------
53  -- Ruta de datos
54  ------------------------------------------------------------
55
56    -- Registro de desplazamiento para el número binario
57    RegDesplBin : process (clk, reset_n)
58    begin  -- process RegDesplBin
59      if reset_n = '0' then
60        reg_bin <= (others => '0');
61      elsif clk'event and clk = '1' then
```

```
62        if crg_bin = '1' then
63          reg_bin <= unsigned(n_bin);
64        elsif despl = '1' then                -- Despl. a izq.
65          reg_bin(7 downto 1) <= reg_bin(6 downto 0);
66          reg_bin(0)          <= '0'; -- Se rellena con ceros.
67        end if;
68      end if;
69    end process RegDesplBin;
70
71    -- Registros de desplazamiento para los dígitos binarios
72
73    -- Registro para las unidades
74    RegDesplBin2BCD_1: RegDesplBin2BCD
75      port map (
76        clk     => clk,
77        reset_n => reset_n,
78        es      => reg_bin(7),
79        despl   => despl,
80        suma_3  => suma_3,
81        pst_a_0 => puesta_a_0,
82        n_bcd   => u_bcd,
83        ss      => u_ss);
84
85    -- Registro para las decenas
86    RegDesplBin2BCD_2: RegDesplBin2BCD
87      port map (
88        clk     => clk,
89        reset_n => reset_n,
90        es      => u_ss,
91        despl   => despl,
92        suma_3  => suma_3,
93        pst_a_0 => puesta_a_0,
94        n_bcd   => d_bcd,
95        ss      => d_ss);
96
97    -- Registro para las centenas
98    RegDesplBin2BCD_3: RegDesplBin2BCD
99      port map (
100       clk     => clk,
101       reset_n => reset_n,
102       es      => d_ss,
103       despl   => despl,
104       suma_3  => suma_3,
105       pst_a_0 => puesta_a_0,
```

```
106       n_bcd   => c_bcd,
107       ss      => open);-- La última salida serie se descarta
108
109   -- Contador para el número de bits procesados (8)
110   CntBits: process (clk, reset_n)
111   begin  -- process CntBits
112     if reset_n = '0' then
113       cnt_bits <= (others => '0');
114     elsif clk'event and clk = '1' then
115       if puesta_a_0 = '1' then
116         cnt_bits <= (others => '0');
117       elsif en_cnt = '1' then
118         cnt_bits <= cnt_bits + 1;
119       end if;
120     end if;
121   end process CntBits;
122   fin_cnt <= '1' when cnt_bits = 7 else
123              '0';
124 ------------------------------------------------------------
125 -- Circuito de control
126 ------------------------------------------------------------
127
128   VarEstado : process (clk, reset_n)
129   begin
130     if reset_n = '0' then               -- Reset asíncrono
131       estado_act <= Reposo;             -- (activo bajo)
132     elsif clk'event and clk = '1' then  -- Flanco de subida
133       estado_act <= estado_sig;
134     end if;
135   end process VarEstado;
136
137   TransicionEstados : process (estado_act, conv, fin_cnt)
138   begin
139     -- Por defecto nos quedamos en el estado actual
140     estado_sig <= estado_act;
141     case estado_act is
142       when Reposo =>
143         if conv = '1' then
144           estado_sig <= Carga;
145         end if;
146       when Carga =>
147         estado_sig <= Desplaza;
148       when Desplaza =>
149         if fin_cnt = '1' then
```

```
150           estado_sig <= FinConv;
151         else
152           estado_sig <= Suma3;
153         end if;
154       when Suma3 =>
155         estado_sig <= Desplaza;
156       when FinConv =>
157         if conv = '0' then -- Evita conversiones seguidas
158                            -- si conv dura mucho tiempo a 1.
159           estado_sig <= Reposo;
160         end if;
161       when others =>
162         estado_sig <= Reposo;
163     end case;
164   end process;
165
166   Salidas: process (estado_act)
167   begin  -- process Salidas
168     crg_bin <= '0';
169     puesta_a_0 <= '0';
170     despl <= '0';
171     en_cnt <= '0';
172     suma_3 <= '0';
173     hecho <= '0';
174     case estado_act is
175       when Reposo =>
176         null;
177       when Carga =>
178         crg_bin <= '1';
179         puesta_a_0 <= '1';
180       when Desplaza =>
181         despl <= '1';
182         en_cnt <= '1';
183       when Suma3 =>
184         suma_3 <= '1';
185       when FinConv =>
186         hecho <= '1';
187       when others => null;
188     end case;
189   end process Salidas;
190
191 end behavioral;
```

Una simulación del circuito se muestra en la figura 11.11.

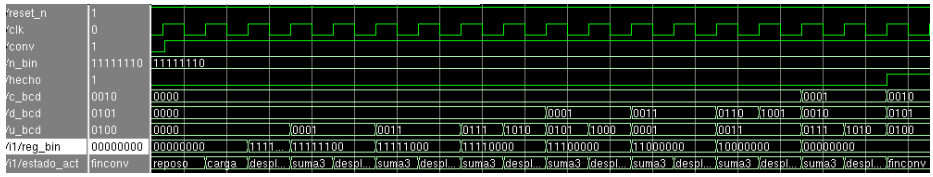


Figura 11.11: Simulación del conversor de binario a BCD.

11.5. Interconexión de dispositivos mediante SPI

En los sistemas basados en microprocesador es frecuente usar interfaces serie para intercomunicar distintos dispositivos. Estos interfaces permiten simplificar la interconexión de dichos dispositivos, ya que usan sólo unos pocos cables para enviar los datos en serie. Un interfaz muy usado es el denominado *Serial Peripheral Interface* (SPI), el cual vamos a estudiar en esta sección. Para ello vamos a diseñar un dispositivo emisor, denominado maestro, y uno receptor, denominado esclavo.

El bus consta tan solo de 4 líneas:

- `sclk`. Reloj para sincronizar las transferencias de datos. Ha de generarla el maestro respetando la frecuencia máxima admitida por el esclavo, que suele estar entre 1 y 70 MHz.
- `ce` (*Chip Enable*). Es una línea para seleccionar el dispositivo al que se desean enviar los datos.[3] Esta línea permanece en reposo a nivel alto y pasa a nivel bajo durante la transmisión de datos.
- `mosi` (*Master Out Slave In*). Salida de datos desde el maestro hacia el esclavo.
- `miso` (*Master In Slave Out*). Entrada de datos provenientes del esclavo hacia el maestro.

La transmisión de datos es *full duplex*, lo que traducido al román paladino quiere decir que se puede enviar un dato y recibir otro al mismo tiempo. Esto es posible gracias a la existencia de dos líneas (`miso` y `mosi`) para transmitir datos. Existen otros interfaces como el I^2C (*Inter-Integrated Circuit*) que sólo disponen de una línea para enviar y recibir datos, por lo que en un instante determinado sólo se puede o enviar o recibir un dato y por ello se les denomina *half duplex*.

La transmisión de datos se realiza formando un anillo entre dos registros de desplazamiento, uno situado en el maestro y otro en el esclavo, tal como se ilustra en la figura 11.12. El número de bits de estos registros no está especificado por el estándar,[4] aunque el más frecuente es 8 bits. El funcionamiento de una transmisión

[3] Si se quiere compartir el bus por varios dispositivos es necesario usar una línea de selección para cada uno de ellos, de forma que se pueda elegir a cuál de los dispositivos va dirigida la información.

[4] El interfaz SPI es lo que se conoce como un estándar de facto, lo cual quiere decir que no hay un organismo detrás que vigile que todos los dispositivos cumplan una normativa exacta. Esto origina que existan ciertas variantes al protocolo, una de las cuales es el tamaño del dato enviado.

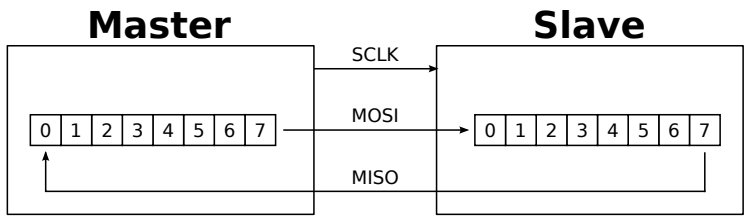


Figura 11.12: Intercambio de datos en el interfaz SPI.

es el siguiente: antes de comenzar maestro y esclavo escriben el dato a transmitir en sus registros de desplazamiento. Cada ciclo de reloj se desplaza un dato desde el maestro al esclavo y otro desde el esclavo al maestro, comenzando con el bit más significativo. Cuando se hayan producido 8 ciclos, los datos presentes en el maestro y en el esclavo se habrán intercambiado, pudiendo entonces repetir el ciclo para intercambiar otro byte.

En la figura 11.13 se muestra un diagrama de tiempos de la transmisión de un byte por un interfaz SPI. El envío comienza con la bajada de la señal `ce_n`. En este momento el maestro situará el primer bit en la salida `mosi` y el esclavo hará lo mismo en la señal `miso`. En cada flanco de bajada de reloj se volverán a desplazar los datos, apareciendo en `mosi` y `miso` dos nuevos datos. Como el reloj se transmite entre dos dispositivos, para evitar problemas si éste se retrasa, los datos se muestrean en los flancos de subida del reloj. De esta manera tienen un semi-periodo para estabilizarse desde que son desplazados por el emisor hasta que son leídos por el receptor.

Por último, destacar que aunque el interfaz es *full duplex* y que el hardware siempre intercambia el contenido de los registros de desplazamiento, en ocasiones la transmisión sólo tiene sentido en un sentido, descartándose los datos recibidos por la otra línea. Por ejemplo, en una memoria flash la comunicación consiste en el envío en primer lugar de la dirección de memoria que se quiere leer desde el maestro hacia el esclavo. Mientras se está enviando esa dirección el esclavo no

Aunque la mayoría de dispositivos trabajan con transmisiones de 8 bits, existen dispositivos que trabajan con otros tamaños.

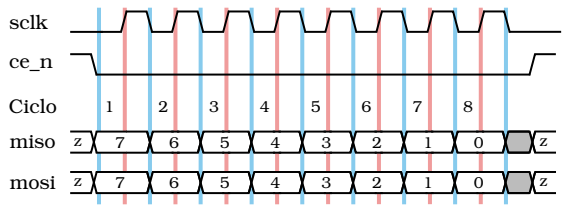


Figura 11.13: Diagrama de tiempo de una ráfaga SPI.

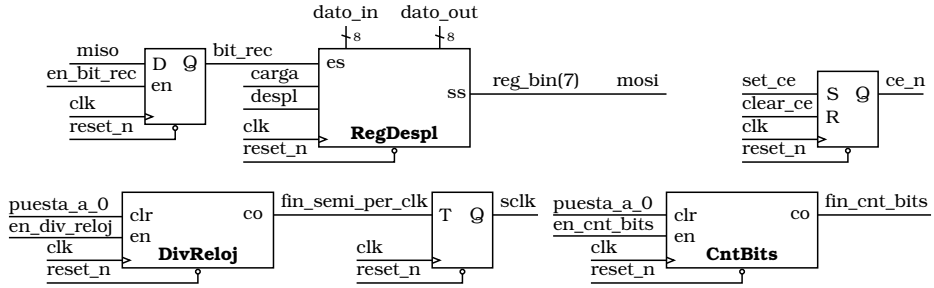


Figura 11.14: Ruta de datos del maestro SPI.

responde nada, por lo que el maestro ha de descartar los bytes recibidos durante el envío de la dirección. De la misma forma, para que el esclavo envíe el dato al maestro, éste ha de enviarle algo, aunque sea basura que el esclavo se encargará de descartar.

11.5.1. Ruta de datos

La ruta de datos del circuito se muestra en la figura 11.14 y consta de los siguientes elementos:

- Un contador de 3 bits (CntBits) que lleva la cuenta del número de bits enviados, activando la señal `fin_cnt_bits` cuando se han enviado los 8 bits. Lo pone a cero el circuito de control al inicio de la transmisión y lo incrementa cada vez que se desplaza un bit, para lo que se usan las señales `puesta_a_0` y `en_cnt_bits`.
- Un divisor de reloj (DivReloj) que genera un acarreo cada vez que ha transcurrido un semiperiodo del reloj SPI. Esta señal se usa para invertir la salida `sclk` y también por la máquina de estados para muestrear y desplazar los datos. Al igual que el contador anterior, la puesta a cero y la habilitación de la cuenta los controla la máquina de estados.
- Un *flip-flop* tipo D que permite muestrear el dato de entrada (`miso`) en el flanco de subida del reloj SPI. Este muestreo lo activa la máquina de estados mediante la señal `en_bit_rec`.
- Un registro de desplazamiento (RegDespl) que sirve para convertir los datos de paralelo a serie y viceversa. El registro cuenta con una entrada serie que se conecta a la salida del *flip-flop* anterior, una salida serie que se conecta a la señal `mosi`, una entrada paralelo por la que se carga el dato a enviar y una salida paralelo por la que se obtiene el dato almacenado en el registro, que al finalizar la transmisión contendrá el dato recibido desde el esclavo.
- Un *flip-flop* tipo RS síncrono para gobernar la señal `ce_n`. Aunque esta señal se podría generar como una salida directa de la máquina de estados, como al

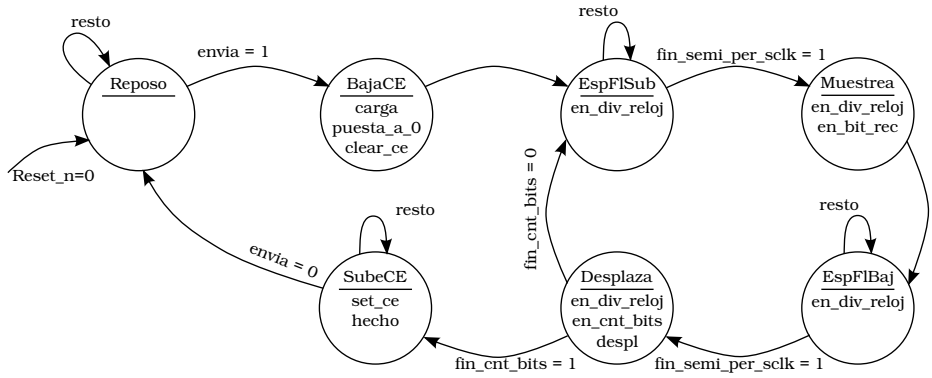


Figura 11.15: Diagrama de estados del maestro SPI.

estar generada mediante un circuito combinacional a partir del estado actual contendría parpadeos que pueden perturbar al receptor. Para evitarlo se usa un *flip-flop* intermedio que se pondrá a cero al inicio de la transmisión y a uno cuando finalice.

11.5.2. Circuito de control

El diagrama de estados del circuito de control se muestra en la figura 11.15. El circuito espera en el estado de reposo la activación de la entrada `envia` que es la que arranca la transmisión del dato. Cuando se active esta entrada se pasa al estado BajaCE, en el que se carga el dato de entrada (`dato_in`) en el registro de desplazamiento y se baja la señal `ce_n` que indica al esclavo el comienzo de la trama. También se aprovecha este estado para poner a cero el contador de bits enviados y el divisor de reloj. De este estado se pasa a EspFlSub, donde se espera la finalización del semiperiodo del reloj del SPI `sclk`, indicado por la salida del divisor de reloj `fin_semi_per_sclk`. En dicho estado y en los cuatro siguientes se activa la señal `en_div_reloj` para mantener activo el divisor de reloj, el cual recordemos que también genera la salida `sclk`. Cuando finalice el semiperiodo se producirá el flanco de subida del reloj SPI y se pasará al estado Muestrea, donde se habilita el *flip-flop* D que almacena el bit presente en la entrada `miso`. De este estado se pasa a EspFlBaj donde se espera otro semiperiodo hasta que baje el reloj `sclk`, pasando entonces al estado Desplaza donde se desplaza el registro de desplazamiento para que aparezca el siguiente bit en la salida `mosi` y se incrementa la cuenta del número de bits transmitidos. De este estado se volverá a EspFlSub a esperar el siguiente flanco de subida si aún no se han enviado todos los bits, o al estado SubeCE si se activa la señal `fin_cnt_bits` para indicar que ya se han enviado todos los bits. Por último, en el estado SubeCE se pone a 1 la señal `ce_n` para indicar el final de la trama al esclavo y la señal `hecho` para indicar el final de la transmisión y que el dato recibido del esclavo puede leerse ya desde la salida `dato_out`. Para evitar enviar el

dato varias veces si la señal envia está activa demasiado tiempo, en este estado se espera a que dicha señal pase a cero antes de volver al estado de reposo.

11.5.3. Descripción en VHDL

A continuación se muestra el código en VHDL del circuito diseñado. Como puede apreciar el código es un reflejo de la ruta de datos y del circuito de control mostrados en las secciones anteriores.

```
1  -- Maestro de un interfaz SPI. Se envían los datos en
2  -- tramas de 8 bits.
3
4  library ieee;
5  use ieee.std_logic_1164.all;
6  use ieee.numeric_std.all;
7
8  entity MaestroSPI is
9
10   port (
11     clk     : in  std_logic;
12     reset_n : in  std_logic;
13     dato_in : in  std_logic_vector(7 downto 0);
14     dato_out: out std_logic_vector(7 downto 0);
15     envia   : in  std_logic;
16     hecho   : out std_logic;
17     ce_n    : out std_logic;
18     sclk    : out std_logic;
19     miso    : in  std_logic;
20     mosi    : out std_logic);
21
22 end MaestroSPI;
23
24 architecture behavioral of MaestroSPI is
25   -- Señales para el registro de desplazamiento
26   signal reg_despl : std_logic_vector(7 downto 0);
27   signal despl : std_logic;
28   signal carga : std_logic;
29   -- Señales para el flip-flop D que muestrea el dato
30   signal bit_rec : std_logic;
31   signal en_bit_rec : std_logic;   -- enable del FF
32   -- Señales para el contador de bits enviados
33   signal cnt_bits : unsigned(2 downto 0);
34   signal en_cnt_bits : std_logic;        -- Enable del cnt
35   signal fin_cnt_bits : std_logic;       -- Fin de cuenta (7)
36   -- Señales para el divisor de reloj que genera sclk
```

```
37    signal div_reloj : unsigned(7 downto 0);
38    constant c_semi_per_sclk : unsigned(7 downto 0) :=
39                               to_unsigned(10,8);
40    signal en_div_reloj : std_logic;
41    signal fin_semi_per_sclk : std_logic;-- Fin del semiperiodo
42    signal sclk_i : std_logic;
43    -- Pone a cero cnt_bits y div_reloj y el flip-flop T
44    -- de salida de sclk
45    signal puesta_a_0 : std_logic;
46    -- Señales para controlar el FF RS síncrono de la señal CE
47    signal set_ce, clear_ce : std_logic;
48    -- Estados
49    type t_estados is (Reposo, BajaCE, EspFlSub, Muestrea,
50                       EspFlBaj, Desplaza, SubeCE);
51    signal estado_act, estado_sig : t_estados;
52  begin  -- behavioral
53
54  ------------------------------------------------------------
55  -- Ruta de datos
56  ------------------------------------------------------------
57
58    -- Contador del número de bits enviados
59    CntBits: process (clk, reset_n)
60    begin  -- process CntBits
61      if reset_n = '0' then
62        cnt_bits <= (others => '0');
63      elsif clk'event and clk = '1' then
64        if puesta_a_0 = '1' then
65          cnt_bits <= (others => '0');
66        elsif en_cnt_bits = '1' then
67          cnt_bits <= cnt_bits + 1;
68        end if;
69      end if;
70    end process CntBits;
71    fin_cnt_bits <= '1' when cnt_bits = 7 else
72                    '0';
73
74    -- Divisor de reloj y FF tipo T para generar sclk
75    DivReloj: process (clk, reset_n)
76    begin  -- process DivReloj
77      if reset_n = '0' then
78        div_reloj <= (others => '0');
79        sclk_i <= '0';
80      elsif clk'event and clk = '1' then
```

```
81        if en_div_reloj = '0' then
82          div_reloj <= (others => '0');
83          sclk_i <= '0';
84        else
85          if div_reloj = c_semi_per_sclk then
86            div_reloj <= (others => '0');
87            sclk_i <= not sclk_i;
88          else
89            div_reloj <= div_reloj + 1;
90          end if;
91        end if;
92      end if;
93    end process DivReloj;
94    fin_semi_per_sclk <= '1' when div_reloj = c_semi_per_sclk
95                         else '0';
96
97    -- Registro desplazamiento para envío y recepción del dato
98    RegDespl : process (clk, reset_n)
99    begin  -- process RegDespl
100     if reset_n = '0' then
101       reg_despl <= (others => '0');
102     elsif clk'event and clk = '1' then
103       if carga = '1' then
104         reg_despl <= dato_in;
105       elsif despl = '1' then               -- Despl. a izq.
106         reg_despl(7 downto 1) <= reg_despl(6 downto 0);
107         reg_despl(0)          <= bit_rec;
108       end if;
109     end if;
110   end process RegDespl;
111
112   -- Flip-flop D para muestrear el dato recibido
113   FFD: process (clk, reset_n)
114   begin  -- process FFD
115     if reset_n = '0' then
116       bit_rec <= '0';
117     elsif clk'event and clk = '1' then
118       if en_bit_rec = '1' then
119         bit_rec <= miso; -- Se muestrea la entrada miso
120       end if;
121     end if;
122   end process FFD;
123
124   -- Flip-flop RS síncrono para generar CE
```

```
125   FFRS: process (clk, reset_n)
126   begin  -- process FFRS
127     if reset_n = '0' then
128       ce_n <= '1';          -- ce_n inactiva tras reset
129     elsif clk'event and clk = '1' then
130       if set_ce = '1' and clear_ce = '0' then
131         ce_n <= '1';
132       elsif set_ce = '0' and clear_ce = '1' then
133         ce_n <= '0';
134       end if;
135     end if;
136   end process FFRS;
137
138   --Asignación de las salidas
139   sclk <= sclk_i;
140   mosi <= reg_despl(7);
141   dato_out <= reg_despl;
142
143 ------------------------------------------------------------
144 -- Circuito de control
145 ------------------------------------------------------------
146   VarEstado : process (clk, reset_n)
147   begin
148     if reset_n = '0' then               -- Reset asíncrono
149       estado_act <= Reposo;             -- (activo bajo)
150     elsif clk'event and clk = '1' then  -- Flanco de subida
151       estado_act <= estado_sig;
152     end if;
153   end process VarEstado;
154
155   TransicionEstados : process (estado_act, envia,
156                                fin_semi_per_sclk,
157                                fin_cnt_bits)
158   begin
159     -- Por defecto nos quedamos en el estado actual
160     estado_sig <= estado_act;
161     case estado_act is
162       when Reposo =>
163         if envia = '1' then
164           estado_sig <= BajaCE;
165         end if;
166       when BajaCE =>
167         estado_sig <= EspFlSub;
168       when EspFlSub =>
```

```
169         if fin_semi_per_sclk = '1' then
170           estado_sig <= Muestrea;
171         end if;
172       when Muestrea =>
173         estado_sig <= EspFlBaj;
174       when EspFlBaj =>
175         if fin_semi_per_sclk = '1' then
176           estado_sig <= Desplaza;
177         end if;
178       when Desplaza =>
179         if fin_cnt_bits = '1' then
180           estado_sig <= SubeCE;
181         else
182           estado_sig <= EspFlSub;
183         end if;
184       when SubeCE =>
185         if envia = '0' then -- Por si envía está aún activo
186           estado_sig <= Reposo;
187         end if;
188       when others =>
189         estado_sig <= Reposo;
190     end case;
191   end process;
192
193   Salidas: process (estado_act)
194   begin
195     despl <= '0';
196     carga <= '0';
197     en_bit_rec <= '0';
198     en_cnt_bits <= '0';
199     en_div_reloj <= '0';
200     puesta_a_0 <= '0';
201     set_ce <= '0';
202     clear_ce <= '0';
203     hecho <= '0';
204     case estado_act is
205       when Reposo => null;
206       when BajaCE =>
207         carga <= '1';
208         puesta_a_0 <= '1';
209         clear_ce <= '1';
210       when EspFlSub =>
211         en_div_reloj <= '1';
212       when Muestrea =>
```

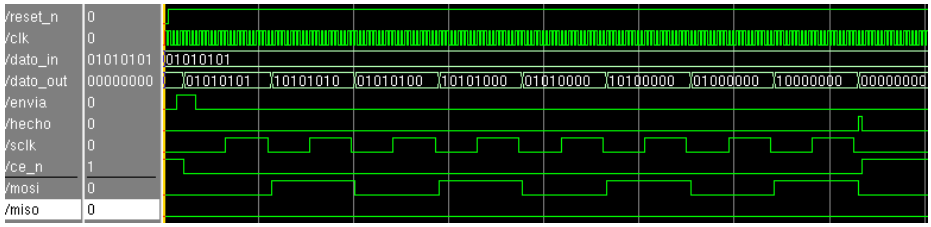


Figura 11.16: Simulación del maestro SPI.

```
213         en_div_reloj <= '1';
214         en_bit_rec <= '1';
215       when EspFlBaj =>
216         en_div_reloj <= '1';
217       when Desplaza =>
218         en_div_reloj <= '1';
219         en_cnt_bits <= '1';
220         despl <= '1';
221       when SubeCE =>
222         set_ce <= '1';
223         hecho <= '1';
224       when others => null;
225     end case;
226   end process Salidas;
227
228 end behavioral;
```

Una simulación del circuito se muestra en la figura 11.16.

Realice el ejercicio 4

11.6. Ejercicios

1. La barrera automática para aparcamiento diseñada en la sección 11.2 falla cuando estando abierta se da un reset. Indiqué en qué consiste el fallo y cómo arreglarlo.
2. Modifique la ruta de datos del circuito para controlar la calidad de los toros diseñado en la sección 11.3 para que use un único temporizador y un comparador de 27 bits. Realice la especificación en VHDL y sintetícela en Quartus, comparando el número de celdas lógicas usadas por este circuito con las usadas por el circuito diseñado en la sección 11.3.
3. Modifique el circuito diseñado en la sección 11.4 para convertir números de 12 bits de binario a BCD. Realice primero un diagrama del camino de datos y justifique si es necesario cambiar el circuito de control. Especifique por último el circuito en VHDL.

4. A partir del maestro SPI diseñado en la sección 11.5, diseñe un esclavo SPI. Para ello tenga en cuenta lo siguiente:
   - Al igual que el maestro dispondrá de una señal de entrada paralelo (`dato_in`) y de otra señal de salida paralelo (`dato_out`).
   - Las señales de reloj `sclk` y de habilitación `ce_n` serán ahora entradas.
   - La señal `miso` será ahora una salida y la señal `mosi` será una entrada.
   - Cuando la señal `ce_n` se ponga a cero cargará el dato presente en la entrada paralelo `dato_in` en el registro de desplazamiento.
   - El esclavo no necesita generar el reloj, limitándose a muestrear el dato cuando llegue un flanco se subida y a desplazarlo cuando llegue un flanco de bajada.
   - El circuito dispondrá de una salida `fin_tr` para indicar que ha terminado una transmisión correcta y que por tanto (si procede) habrá un dato disponible en la salida `dato_out`.
   - La señal `fin_tr` se activará cuando se hayan recibido los 8 bits.
   - Si antes de haber recibido los 8 bits se desactiva la señal `ce_n` es porque ha ocurrido algún error en la transmisión y por tanto no hay que tener en cuenta ninguno de los bits recibidos. Se pasará entonces al estado de reposo sin activar la señal `fin_tr`.

Bibliografía

[Augarten, 1983] Augarten, S. (1983). *State of the Art. A photographic history of the integrated circuit*. Tichnor & Fields, New Haven and New York, 1 edition.

[Boole, 1854] Boole, G. (1854). An investigation of the laws of thought. `http://www.archive.org/details/investigationofl00boolrich`. [Online; último acceso 13 de Febrero de 2011].

[Chu-Carroll, 2006] Chu-Carroll, M. C. (2006). Good math, bad math : Roman numerals and arithmetic. `http://scienceblogs.com/goodmath/2006/08/roman_numerals_and_arithmetic.php`. [Online; último acceso 10 de Mayo de 2010].

[Goldstine and Goldstine, 1996] Goldstine, H. and Goldstine, A. (1996). The electronic numerical integrator and computer (ENIAC). *Annals of the History of Computing, IEEE*, 18(1):10–16.

[Intel, 1973] Intel (1973). 4004 schematics. `http://www.intel.com/about/companyinfo/museum/exhibits/4004/docs.htm`. [Online; último acceso 10 de Febrero de 2011].

[Marks II, 1991] Marks II, R. J. (1991). *Introduction to Shannon Sampling and Interpolation Theory*. Springer-Verlag.

[Van der Spiegel, 1995] Van der Spiegel, J. (1995). Eniac-on-a-Chip project web page. `http://www.ese.upenn.edu/~jan/eniacproj.html`. [Online; último acceso 20 de Enero de 2011].

[Wikipedia, 2010] Wikipedia (2010). Maya numerals — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Maya_numerals&oldid=358828137`. [Online; último acceso 10 de Mayo de 2010].

[Wikipedia, 2011] Wikipedia (2011). Apollo guidance computer — Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Apollo_Guidance_Computer`. [Online; último acceso 5 de Febrero de 2011].

Índice alfabético

bit de signo, 47, 49

error de cuantización, 40
exceso, 51
extensión de signo, 54

sesgo, 51